Application d'un langage de programmation de type flot de données à la synthèse haut-niveau de système de vision en temps-réel sur matériel reconfigurable

Sameer Ahmed

N° d'ordre : D.U.2334
EDSPIC : 605

Université Blaise Pascal - Clermont-Ferrand II

École Doctorale
Sciences pour l'Ingénieur de Clermont-Ferrand

Thèse

présentée par

Sameer Ahmed

pour obtenir le grade de

Docteur d'Université
Spécialité: Vision pour la Robotique

Application of a Dataflow Programming Language to the High Level Synthesis of Real-Time Vision Systems on Reconfigurable Hardware

Soutenue publiquement le 24 Janvier 2013 devant le jury:

M. Greg Michaelson	Rapporteur
M. Hassan Rabah	Rapporteur
M. Dominique Ginhac	Examinateur
M. Julien Dubois	Examinateur
M. Jean Pierre Derutin	Examinateur
M. François Berry	Examinateur
M. Jocelyn Sérot	Directeur de thèse

Abstract

Field Programmable Gate Arrays (FPGAs) are reconfigurable devices which can outperform *General Purpose Processors* (GPPs) for applications exhibiting parallelism. Traditionally, FPGAs are programmed using *Hardware Description Languages* (HDLs) such as Verilog and VHDL. Using these languages generally offers the best performances but the programmer must be familiar with digital design. This creates a barrier for the software community to use FPGAs and limits their adoption as a computing solution.

To make FPGAs accessible to both software and hardware programmers, a number of tools have been proposed both by academia and industry providing high-level programming environment. A widely used approach is to convert C-like languages to HDLs, making it easier for software programmers to use FPGAs. But these approaches generally do not provide performances on the par with those obtained with HDL languages. The primary reason is the inability of C-like approaches to express parallelism. Our claim is that in order to have a high level programming language for FPGAs as well as not to compromise on performance, a shift in programming paradigm is required. We think that the *dataflow/actor* programming model is a good candidate for this.

This thesis explores the adoption of *dataflow/actor* programming model for programming FPGAs. More precisely, we assess the suitability of CAPH, a domain-specific language based on this programming model for the description and implementation of stream-processing applications on FPGAs. The expressivity of the language and the efficiency of the generated code are assessed experimentally using a set of test bench applications ranging from very simple applications (basic image filtering) to more complex realistic applications such as motion detection, Connected Component Labeling (CCL) and JPEG encoder.

Keywords: Dataflow programming, stream-processing applications, FPGA, computer vision.

Résumé

Les circuits reconfigurables de type FPGA *(Field Programmable Gate Arrays)* peuvent désormais surpasser les processeurs généralistes pour certaines applications offrant un fort degré de parallélisme intrinsèque. Ces circuits sont traditionnellement programmés en utilisant des langages de type HDL (*Hardware Description Languages*), comme Verilog et VHDL. L'usage de ces langages permet d'exploiter au mieux les performances offertes par ces circuits mais requiert des programmeurs une très bonne connaissance des techniques de conception numérique. Ce pré-requis limite fortement l'utilisation des FPGA par la communauté des concepteurs de logiciel en général.

Afin de pallier cette limitation, un certain nombre d'outils de plus haut niveau ont été développés, tant dans le monde industriel qu'académique. Parmi les approches proposées, celles fondées sur une transformation plus ou moins automatique de langages de type C ou équivalent, largement utilisés dans le domaine logiciel, ont été les plus explorées. Malheureusement, ces approches ne permettent pas, en général, d'obtenir des performances comparables à celles issues d'une formulation directe avec un langage de type HDL, en raison, essentiellement, de l'incapacité de ces langages à exprimer le parallélisme intrinsèque des applications. Une solution possible à ce problème passe par un changement du modèle de programmation même. Dans le contexte qui est le notre, le modèle *flot de données* apparaît comme un bon candidat.

Cette thèse explore donc l'adoption d'un modèle de programmation flot de données pour la programmation de circuits de type FPGA. Plus précisémment, nous évaluons l'adéquation de CAPH, un langage orienté domaine (*Domain Specific Language*) à la description et à l'implantation sur FPGA d'application opérant à la volée des capteurs (*stream processing applications*). L'expressivité du langage et l'efficacité du code généré sont évaluées expérimentalement en utilisant un large spectre d'applications, allant du traitement d'images bas niveau (filtrage, convolution) à des applications de complexité réaliste telles que la détection de mouvement, l'étiquetage en composantes connexes ou l'encodage JPEG.

Mots-clefs: Modèle flot de données, FPGA, traitement d'images, vision par ordinateur.

Dedicated to my father

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are reconfigurable devices used for implementation of digital logic circuits. In the past decade, there has been a tremendous increase in the capacity of FPGAs. Moreover, many applications exhibiting parallelism, when implemented on FPGAs can outperform *General Purpose Processors* (GPPs). An example of such a class of applications is stream-processing applications. These applications operate on continuous streams of data and require high computing power. Furthermore, most of the computationally demanding tasks in these applications show parallelism. This makes FPGAs, a good candidate for implementing these applications.

FPGAs are programmed with *Hardware Description Languages* (HDLs) such as Verilog and VHDL. These languages provide the best performances in terms of area and speed. But, since these languages were designed for hardware designers, one has to acquire expertise in digital design to use them. From a programming point of view, this means that the FPGA programming community is limited to hardware experts. A desirable objective, to enlarge this community -and therefore the use of FPGA- is to make programming accessible to both hardware as well as software programmers.

To make this possible, a lot of tools have been proposed, both from academia and industry to provide high-level programming environment for FPGAs. The most commonly used approach, is to convert C-like languages into (V)HDL. Since C is widely used by the software community, this makes it easier for software programmer to program FPGAs. But this shift comes at a cost. As discussed earlier, HDLs provide best performances which is crucial for large applications. C-like approaches have to compromise on these performances. There are many reasons for this. The primary reason is the incapability of C-like languages to express parallelism in applications, which is the main factor of performance gain on FPGAs. Since C is intrinsically sequential, the task to identify parallelism is left to the compiler. In the current state-of-the-art this cannot be fully accomplished in an automatic way.

This means that in order to have a high level programing model for FPGA as well as not to compromise on performance, a shift in programming paradigm is required. In other words, it is crucial to reduce the gap between the *programming* model (as viewed by the programmer) and *implementation* model (as implemented in target hardware). The *Dataflow/actor* programming models, seem to be good candidate for this.

This thesis explores the adoption of a *dataflow/actor* programming model for programming stream-processing applications on FPGAs. More precisely, it investigates whether applications can be implemented at higher level using this model without compromising on performance. CAPH [3], a domain specific language based on the *dataflow/actor* programming model, is used to evaluate the aforementioned. The development and implementation of CAPH was independent of this thesis work. The main contribution of the thesis lies in the experimental benchmark of CAPH. First, a set of simple applications are developed to test whether it is feasible to use CAPH for programming FPGAs, more specifically to test whether it improves programmer efficiency or not. The results reported here show that CAPH can be used to efficiently implement applications for FPGAs at a higher abstraction level. In the second step, the performance of CAPH is evaluated by using more complex applications (Motion Detection Application, Connected Component Labeling (CCL) and JPEG encoder). For the last application results are also compared with direct VHDL implementation as well as another popular dataflow language CAL. The comparison with the former is used to prove the gain in

expressivity offered by a higher level language does not come at the price of a reduced efficiency.

The main research contribution of this thesis is to evaluate CAPH for programming image processing applications on FPGAs, this includes :

- At start, development of simple image processing applications in CAPH.
- Later, a benchmark of complex image processing applications implemented in CAPH.
- The experimental results of one application (JPEG encoder) are compared with handwritten VHDL and another popular dataflow language CAL on two different FPGA platforms.

The overall organization of the thesis is as follows:

Chapter 2 starts by introducing FPGAs, their architecture and the reason for their current emergence in the reconfigurable computing domain. Programming issues, which are the main obstacle to their widespread acceptance are described. Several state of the art C-like approaches to this problem are described and the reason why these approaches fail to meet the required performance are explained. Then, the *Dataflow/actor* programming model is proposed as an alternative programming model and a introduction to the dataflow programming model is given. Since this model was initially designed to program dataflow machines, some of the earlier and most famous languages based on this model to program dataflow machines are described. The recent and renewed interest in this model is due to the emergence of FPGAs. The reason for the natural coherence of this model for programming FPGAs are outlined. This chapter will also describe some languages based on the dataflow programming model for programming FPGAs.

Chapter 3 gives an overview of the CAPH language. The main constructs of the language are illustrated with examples. Like other dataflow/actor based languages, applications are described at two levels : one to describe the behavior of each actor and the other the interconnections between the actors. The distinguished features of CAPH in describing these two levels, as compared to the other languages discussed in the previous chapter, are also highlighted. We also describe the design flow of the compiler and how this flow is supported by tools offered by the CAPH language.

Chapter 4 discusses the main issues related to the generation of VHDL code from CAPH programs. First of all, the representation of stream tokens is described at the hardware level. The process of adding control tokens to the input stream and the encoding technique used to distinguish them from data tokens is explained. With the help of small examples, an analysis of the VHDL code generated for each CAPH statement/construct is conducted. This covers the code generated for both the actor(s) and network parts. Issues related to FIFOs (used to connect actors) are also discussed, since this is an important aspect of dataflow/actor model.

Chapter 5 focuses on some programming features introduced by CAPH distinct from the basic programming constructs described in chapter 3. This chapter is divided into three parts. The first part focuses on expressing arithmetic operations. The use of built-in library operators provided by CAPH and the implementation of more complex operators are demonstrated. The second part describes some memory-related features, which play an important role in image

processing applications, as used by many applications in the next chapter. The implementation of a feature at the CAPH level as well as the resource utilization of the resulting VHDL design are explained in detail. Finally, the functional features of CAPH are analyzed with the help of examples which help to access their utility in improving application expressivity.

Chapter 6 starts by introducing the target platform we used to test CAPH applications on FPGA. Then it moves to demonstrate the effectiveness of the CAPH language both in terms of expressivity and performance. These are validated by describing the implementation of several test bench applications. These applications include motion detection, connected component labeling and parts of a JPEG encoder. Each application/experiment is described as follows : first an introduction and the main objective of the implemented algorithm are given, then its formulation in CAPH is described and finally, for the target FPGA, both resource consumption (Logic Elements (LEs), memory bits, Digital Signal Processing (DSP) blocks etc.) and performance (max. clock frequency, frame per second (FPS)) are given. For the last application (i.e. the JPEG encoder parts), a comparative analysis is also made with a direct VHDL implementation and another dataflow language CAL.

Chapter 7 concludes the thesis, by highlighting the original ideas explored and giving some directions for future research work.

Chapter 2

Reconfigurable Computing

Reconfigurable computing (RC) refers to the ability for a system to provide some form of hardware reprogrammability. By using RC, the same hardware can be changed to execute different applications [4]. This innovative development of hardware for an unlimited amount of reuse by re-programming led to a new field where many different hardware algorithms can execute on a single hardware, as many different software algorithms can run on a conventional microprocessor. Although the field of reconfigurable computing is not new [5], the recent surge in the field is due to rapid development of FPGAs [6].

2.1 FPGAs

With the increase in application complexity there is a constant need for more computing power, especially for applications like video processing, image recognition and processing etc. The situation becomes more complex, when considering the factors like power consumption, manufacturing cost and time to market. To keep on increasing processing power and decreasing the aforementioned factors is a challenging task. Single high performance microprocessors simply cannot meet the performance requirement for the computationally intensive applications. The above problems with conventional microprocessors have led to the recent interest in FPGAs. The decrease in performance gain of conventional microprocessor, in addition to considerable cost of their power requirement has left a vacuum to be filled by any other cost-effective technology and FPGAs seem to fill this gap.

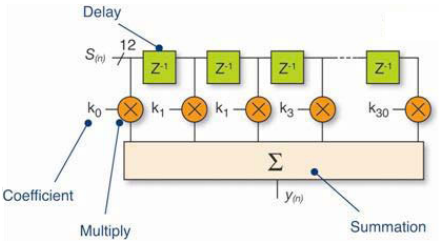


Figure 2.1: Implementation of 32 tap FIR filter on FPGA

In fact, due to their reconfigurable architecture, FPGAs perform hardware optimizations of resources for an application as opposed to traditional processors. This results in the hardware configuration of the FPGA according to the application. On the contrary, the traditional processor has a fixed architecture and relies on a high clock frequency or duplication of processing cores. For example, considering a signal processing algorithm FIR filter for 32 samples, an FPGA performs massive parallelism with a pipeline of 32 registers (Fig. 2.1) and produces output at each clock cycle. On the other hand, implementation on a traditional processor with an arithmetic and logic unit (ALU) will perform 32 iterations to produce the same result (Fig. 2.2).

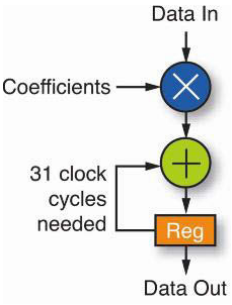


Figure 2.2: The implementation is known as a multiply-and-accumulate or MAC-type implementation. This is almost certainly the way a FIR filter would be implemented on a classical processor.

It is therefore easy to understand that apart from the possibility of reconfiguration of the FPGA hardware, FPGA works in "space" while a traditional processor works in "time". In the next section, we describe the internal structure of an FPGA emphasizing the characteristics of different modules.

2.1.1 FPGA Architecture

FPGAs are components invented by Xilinx in the early 1980s [7] and improve the characteristics of the CPLD (Complex Programmable Logic Device) type circuits. The most common FPGA architecture consists of an array of logic blocks (called Configurable Logic Block (CLB) or Logic Array Block (LAB) depending on the vendor), I/O pads, and routing channels as shown in Fig. 2.3. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array.

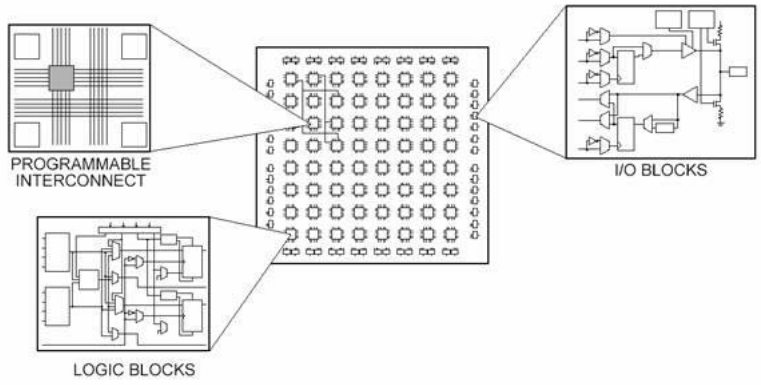


Figure 2.3: Generic FPGA architecture

With the passage of time, the basic architecture of FPGA has evolved to include more specialized programmable logic blocks. These include embedded memory, arithmetic logic (multiplier or Digital Signal Processing (DSP) blocks), high speed I/O and even embedded microprocessors. In the sections below, these main blocks are presented.

2.1.1.1 Logic Block

Since their invention in 1980's, FPGAs have used a large variety of structure for logic block. A simplified architecture of a programmable logic block is shown in Fig. 2.4. It consists of

programmable combinational logic, a flip-flop or latch and carry chain logic. The output of the block is either the output of the combinational logic or the output of the flip-flop. The logic block in commercial FPGAs is much more flexible than this simple one. The most common way to implement the combinational logic is a look-up table (LUT), which acts as a memory with N address lines and 2^N memory locations. In order to implement a specific function, the truth table has to be loaded into the memory. Because of area efficiency, most commercial FPGAs use four-input LUTs. Many FPGAs combine logic blocks to form a *cluster* in order to reduce the cost of routing. A special faster routing named "regional routing" is provided to connect logic blocks inside a cluster. This helps implementing larger functions inside a cluster where routing is the "speed bottleneck". For example, in the Altera Stratix family [8], each logic block consists of 4-input LUTs and 10 logic blocks are combined to form a cluster called Logic Array Block (LAB). Later Altera Stratix families use a mini-cluster known as Adaptive Logic Module (ALM) [9]. Each ALM consists of eight input adaptive look-up table (LUT), two dedicated embedded adders, and four dedicated registers. On the Xilinx side, the mini-cluster is called a Slice. Virtex 6 Slices consist of four look-up tables, eight registers, wide function multiplexer and carry logic [10]. The association of two slices is called a Configurable Logic Block (CLB).

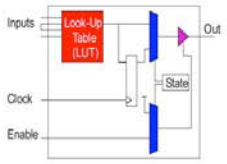


Figure 2.4: Generalized FPGA Logic Element

2.1.1.2 Routing Architecture

Logic Array blocks (LABs), DSP blocks, memory blocks and I/Os need to be connected through routing. Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box (Fig. 2.6). By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

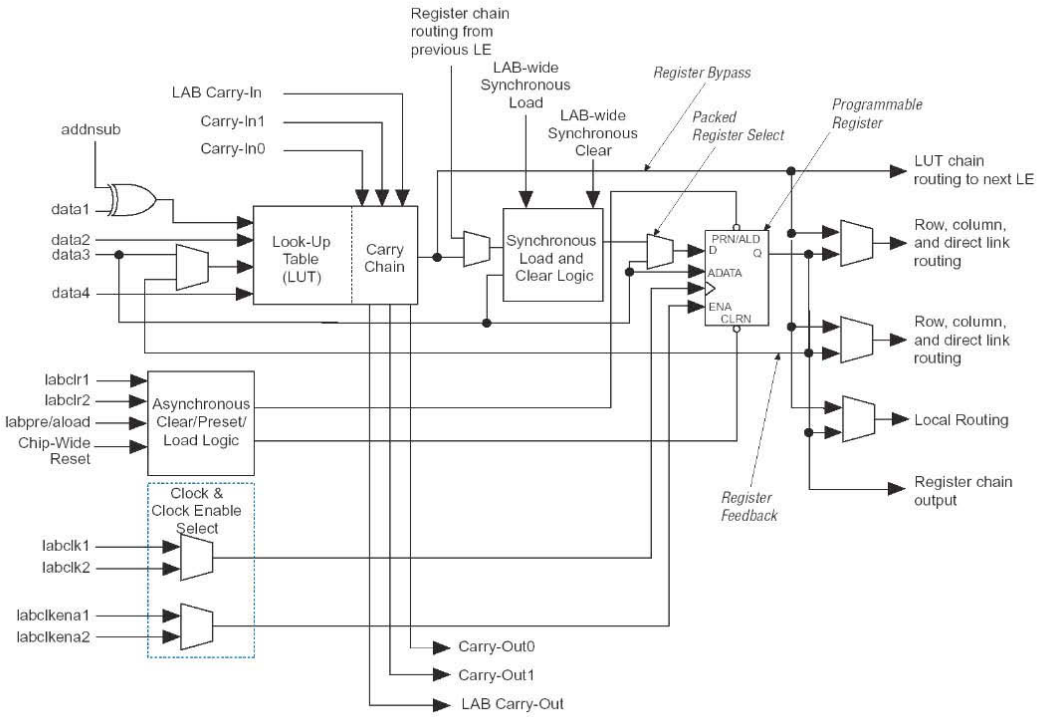


Figure 2.5: Altera Cyclone II Logic Element [1]

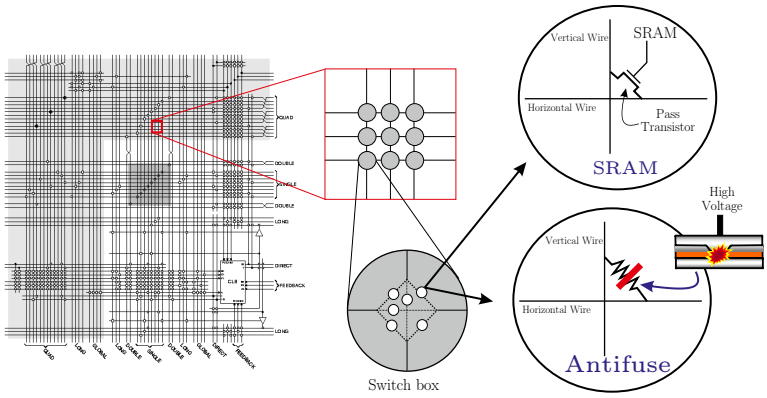


Figure 2.6: FPGA routing technology. There are two main approaches to configure the routing network. First, SRAM-based (Static RAM) where the configuration bitstream is stored in a classical SRAM. Since SRAM is volatile and cannot keep data without a power source, such FPGAs must be programmed (configured) upon startup. The majority of FPGAs use this routing approach. Second is the antifuse-based approach, where each device does not conduct current initially, but can be "burned" to conduct current (the antifuse behavior is thus opposite to that of the fuse, hence the name). The antifuse-based FPGAs cannot be reprogrammed since there is no way to return a burned antifuse into the initial state.

Whenever a vertical and a horizontal channel intersect, there is a switch box. In this architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The pattern or topology of switches used in this architecture is the planar or domain-based switch box topology. In

this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on.

Based on the switch and wire, interconnect routes can be modeled as RC networks (Fig. 2.7). The modeling explains why the placement is often a crucial point in a design. The routing length causes time delays in the path.

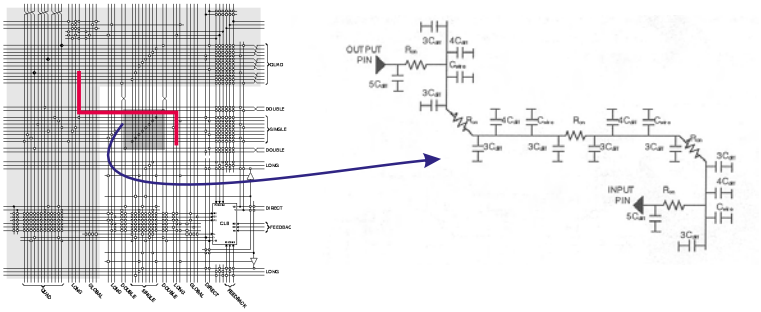


Figure 2.7: FPGA routing modeling

2.1.1.3 Input and Outputs

Input and output blocks are used to connect an FPGA with external devices. Similar to dedicated logic blocks, FPGAs also include dedicated I/O hardware (for example, for DDR (double data rate) memories). In the Altera Stratix device, each I/O pin has an I/O element (IOE) which is located at the end of LAB rows and columns. Each IOE contains a bidirectional I/O buffer and six registers for registering input, output, and output-enable signals. There are also high speed serial interface channels which support up to 840 Mbps transfer rates.

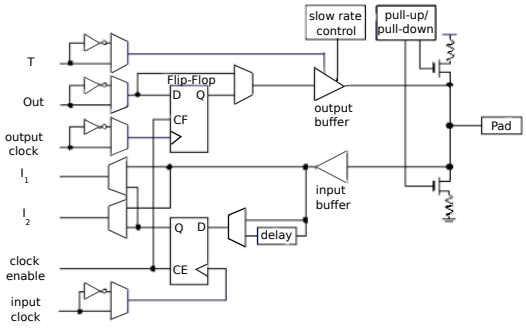


Figure 2.8: Typical I/O pad from Altera Stratix. FPGAs provide support for dozens of I/O standards (TTL, CMOS, LVDS etc.) which are grouped in banks.

2.1.1.4 Others blocks

Apart from the previously described blocks, most of the latest FPGAs also integrate two other dedicated blocks : embedded memory blocks and dedicated arithmetic blocks.

(a)Memory blocks

The circuitry inside logic blocks can be used for memory but they are inefficient for creating memories of large depth. So, FPGA vendors started providing SRAM blocks within the architecture. The classical ways to use memory inside FPGAs are:

– Register File

– Shift-register block

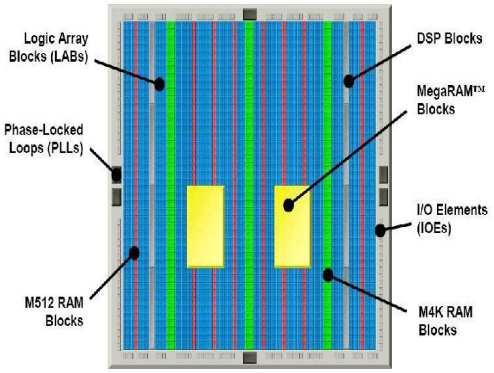
– ROM and waveform generation

– First-in-First-Out memory

In the Altera Stratix, it is a TriMatrix memory (Fig. 2.9, consisting of three type of RAM blocks : M512, M4K and M-RAM blocks). M512 blocks consist of 512 bits plus parity (576 bits). They can be configured with aspect ratio from 512x1 to 32x18. M4K blocks consist of 4K bits plus parity (4,608 bits). They can be configured with aspect ratio from 4Kx1 to 128x36. Finally, the M-RAM blocks consist of 512K bits plus parity (589,824 bits). These blocks can be configured with aspect ratio from 64Kx8 to 4Kx144. The memory sizes of different range facilitate the best size to be selected for the application needs without wasting many resources.

Memory Block	Applications
M512 / MLAB	• Shift registers • Small FIFO buffers • Filter delay lines
M4K / M9K	• General -purpose memory • Packet header or cell buffers
M144K / M-RAM	• Processor code storage • Packet buffers • Video frame buffers

(a)



(b)

Figure 2.9: (a) Typical application for each Stratix memory blocks; (b) Stratix floor-planning [11]

(b)Arithmetic blocks

Logic blocks can be used to perform any operation with the help of carry chain logic and adders but for complex operations it takes more area, delay and power. To overcome this, FPGAs have started including dedicated blocks for arithmetic operations. These blocks can perform addition/subtraction, multiplication and multiply-accumulate (MAC) operations. The Xilinx device consists of 18x18 bit multipliers and Altera device contains DSP blocks. Altera DSP blocks are more flexible than Xilinx multipliers, they can also perform accumulator function along with multiplication. These DSP blocks can be configured to eight 9x9 bit multiplier, four 18x18 multiplier or one 36x36 multiplier. These blocks also contain 18-bit input shift register (Fig. 2.10).

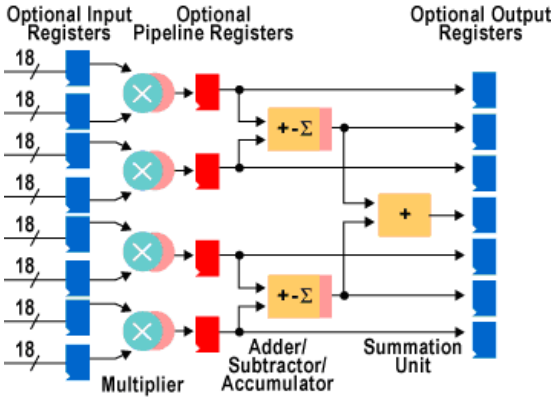


Figure 2.10: Altera Stratix II DSP Block [2]

This section gave an insight into different parts of an FPGA from an architectural perspective along with benefits obtained as compared to traditional processor. The next step is to implement an application to gain the benefits claimed by the device. The next section describes in detail the design flow to implement an application on an FPGA.

2.1.2 Programming FPGAs

To implement a design on FPGA, the design flow consist of several steps as shown in Fig. 2.11. First, the design is described using Hardware Description Languages (HDLs), such as VHDL [12] or Verilog [13]. This hardware description is synthesized and simulated to make sure it gives the intended behavior. The synthesis step takes this description and generates a gate level representation for the FPGA. It actually represents the design in terms of basic building blocks on FPGAs. The output of this design is a netlist[1] in EDIF (electronic design interchange format). Gate level simulation is performed to test that the design is synthesized correctly.

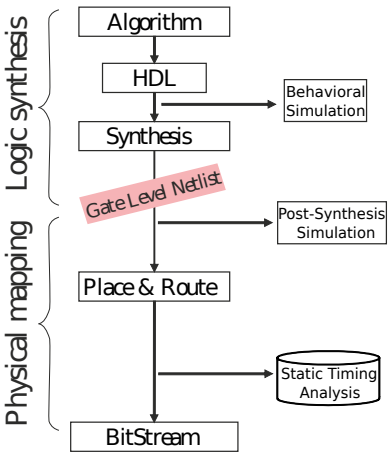


Figure 2.11: FPGA design implementation steps

In the next stage, the synthesized netlist is mapped on to the actual FPGA target. This is accomplished in two steps. In the *mapping* step, the components to perform logic are selected on the FPGA. They consist of selecting one LUT for simple operations or combination of LUTs

[1] A textual description of a circuit diagram

for complex operations. In the *place and route* step, these mapped components are assigned to particular logic blocks on the FPGA and routing is performed to connect these components. This step takes into account the timing requirement for placement and routing of critical paths. In the last step, the configuration file is generated to program the FPGA. Apart from the first step (specifying the design using HDLs), all other steps are performed by CAD tools, usually provided by FPGA vendors. Since a typical user is only directly involved in the first step, we focus on this step in sequel.

Even with modern HDLs such as VHDL [12] or Verilog [13], describing a design is often a daunting task, because it requires a very good knowledge of concepts and techniques which are specific to hardware design. This is a great hurdle to the wide spread acceptance of FPGAs to software programmer community. To overcome this problem, several higher level programming languages have been proposed in recent past. Some of these will be discussed in next section.

2.2 High Level Synthesis (HLS) for FPGAs

The most commonly used source input for high level synthesis is based on standard languages such as ANSI C/C++ [14] and SystemC [15]. In the C-based High-level synthesis languages, the code is analyzed, constrained architecturally, and scheduled to create a register transfer level hardware design language (HDL), which is then synthesized to the gate level by the use of a logic synthesis tool. The goal of HLS is to let software programmers efficiently build and verify hardware design, by giving them better control over optimization of their design architecture. This is achieved by facilitating the programmers to describe the design using higher level tool, where the tool does the RTL implementation. Numerous languages have been proposed by different research teams. In the sequel, some of them are listed:

- Impulse-C [16] by Impulse Accelerated Technologies, is a C-based language for writing applications with the help of a library of functions to describe parallel processes. The communication between processes is based on a stream-based model. Existing VHDL designs can also be used with the help of external functions.

- Handle-C [17] is C-based hardware language provided by Celoxica. It provides statements to define parallel processing elements (`par`) and constructs for communication between them. It also supports flexible width variables, signals and bit-manipulation operations.

- Mitrion-C [18] is also a C-based hardware language by Mitrionics, to write code for FPGA applications. It is a ANSI-C based functional language which means that parallelism is expressed implicitly. The code written in Mitrion-C is converted to code for the Mitrion Virtual Processor (MVP) which is a reconfigurable soft-core processor.

- The Carte-C [19] development environment provides a library of pre-synthesized hardware functions to write programs. Users can also integrate their own VHDL/Verilog macros.

- Stream-C [20], by Los Alamos National Laboratory, is based on the Communicating Sequential Process (CSP) [21] model of computation. It was developed for implementing stream based applications on FPGAs. It consist of annotations for process, stream and signal. A process is independently executing an object consisting of C routines and signals synchronize execution of processes. Streams are used to associate inputs/outputs with each process. With the help of steam information, the compiler generates a process graph.

– SA-C [22] is functional, single assignment language. The compiler generates a dataflow graph of the application before generating FPGA code. SA-C does not include pointers, recursion and while-loops.

– SPARK [23] is high-level language which convert C code to VHDL. It is targeted for multimedia and image processing applications. For computational intensive blocks it performs optimizations such as loop unrolling and code motion to increase instruction level parallelism.

– The DWARV [24] C-to-VHDL generator converts C code into VHDL. The conversion process includes several phases of analysis, transformations and optimizations. It has limited C constructors, which include *if* statements and arithmetic and logic operations over a scalar or one dimensional array of scalar data.

– Mobius [25] is a domain specific,concurrent programming language based on the CSP model of communication. It has a Pascal-like syntax. The processes execute concurrently and exchange data through unidirectional channels.

All of the above languages are C-based except Mobius, nevertheless all expose parallelism by either providing statement-level annotations or relying on compiler to extract parallelism. In the former case, code has to be rewritten, as in case of Stream-C or SA-C. For the later, the compiler has to identify parallelism. In the current state-of-the-art, this cannot be done in a fully automatic way and the programmer is required to put annotations (pragmas) in the code to help the compiler. Finally, the code generally has to undergo various optimizations and transformations before the actual HDL generation. These optimizations and transformations vary from high level parallelization techniques to low level scheduling. The low level optimizations can be beneficial to any algorithm, but the high level optimizations are specifically suggested in the context of one field and would not give performance gains in other domains [24]. Moreover, with some of the existing tools (e.g. Handle-C, Impulse-C), transformations and optimizations require inputs from the programmer [26], who therefore must have a good knowledge of digital design.

All this makes the development of easy to use and efficient programming environments for FPGAs a challenging task [27]. In particular, one can question the use of the C language as a good basis for such an environment. In fact, as C was initially designed for single core architectures, it cannot efficiently be used as a language for expressing parallel computations. Based on this constraint, the research community has started working on domain specific languages (DSLs) for programming FPGAs [28, 29]. By restricting the class of target platforms and embedding some informations which cannot be easily expressed in a more general purpose language, DSLs offer the opportunity to reach an acceptable expressivity versus performance trade off. For this the gap between the programming model (as viewed by the programmer) and execution model (as implemented on the target hardware) must be reduced. The *dataflow* model of computation (MoC) has several properties making this possible. It is described in next section.

2.3 Dataflow Programming

2.3.1 Dataflow Programming Model

The dataflow programming model came into emergence in 1970s with the advent of dataflow architectures which were designed with the objective to exploit massive parallelism [30]. Because of the parallel execution of dataflow programs, these machines were able to overcome the von Neumann architecture bottlenecks [31, 32, 33]. The two major objections to the von Neumann model were the use of a global program counter and global memory [34]. From the start, it was widely acknowledged that imperative languages were not adapted to program machines based on this model [35]. Specific languages – namely dataflow languages– were designed in this context [30, 36]. After the emergence of dataflow architectures in 1970s, the research into the field of *dataflow languages* slowed after mid-1980s. The reason was unavailability of cost-effective dataflow hardware [37].

The name dataflow comes from the conceptual notion that a program in a dataflow computer is a directed graph and that data "flows" between instructions, along its arcs [38]. A program written in dataflow programming language is compiled to a dataflow graph – the "machine language" of dataflow computers [39]. There is no notion of a single point or locus of control - nothing corresponding to the program counter. The nodes of the graph are operators or "instructions". The arrows between the nodes represent data dependencies. Data flows as tokens along the arcs. Incoming arrows that flow toward a node are input to that node and outgoing arrows are output from that node. Whenever a node has all the required data on input, the node is fireable. As a result, it removes the data tokens from input, performs its operation, and places new data tokens on output. It then waits to become fireable again. By this method, nodes are executed as soon as input data becomes available. This stands in contrast to the von Neumann execution model, in which an instruction is only executed when the program counter reaches it, regardless of whether or not it can be executed earlier than this. This model exploits parallelism by executing nodes in a pipeline fashion. A node starts execution as soon as the data is available at the input(s). Fig. 2.12 shows a small program and the corresponding dataflow graph, arrows represent arcs and circles represent instruction nodes. Under the von Neumann execution model, this program would execute sequentially in five steps. In the first step, `a` and `10` are added and result is stored in `k`. In second step, `b` is subtracted from `10` and result is stored in `l`. The third step will multiply `k` with `9` and store result in `m`. Similarly, the fourth step will multiply `l` with `6` and store result in `n`. The last step will divide `m` by `n` and result is stored in `r`. The same program is executed in three steps under the dataflow execution model. The first step performs the addition and subtraction operations simultaneously, as soon as data is available for their execution. Similarly, both multiplication operations are performed in the second step. Finally, in the third step the division operation is performed.

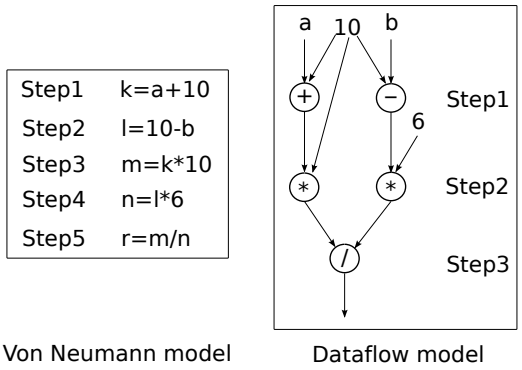


Figure 2.12: Von Neumann vs dataflow execution model

In the Dataflow model, nodes represent units of computation and edges represent FIFO communication channels. By changing the regularity and determinism of the communication pattern, as well as the amount of buffering allowed on the channels, different variants of the dataflow model can be developed. One of the most widely used is Synchronous Dataflow (SDF) [40]. In this model, the numbers of data items produced and consumed by one node at each execution is constant and known at compile time. So, the amount of buffering needed can be determined statically. Many variations of SDF have been defined, including cyclo-static dataflow [41, 42] and multi dimensional synchronous dataflow [43].

Though it was defined before the advent of reconfigurable architectures, the Dataflow MoC appears to be well suited for writing applications targeting FPGAs. In particular, it exploits the inherent concurrency in the algorithm without requiring the programmer to make it explicit. Moreover, the DFG representation of an application is in close resemblance with many image and signal processing algorithms which are represented graphically using block diagrams. This makes dataflow model a natural choice for these applications targeting FPGAs. In [44], authors emphasize the importance of stream architectures and dataflow design techniques to address concurrent design as compared to conventional general purpose languages. These languages are not well suited for representing parallel architectures. One possible solution is to add concurrent constructs but this is not natural and effects readability and programmer productivity. On the other hand, relying completely on the compiler to extract parallelism is not possible in the current state of the art. This advocates for a shift towards the dataflow model. In fact, the recent renewed interest in stream or dataflow programming can be viewed as a consequence of the development of reconfigurable computing (RC)/FPGAs, since this model provides a natural way to program these devices. There are several new languages and the area currently attracts considerable attention from academia and industry. Some of the languages will be introduced in section 2.3.3.

2.3.2 Dataflow Programming Languages

Several programming languages have been designed based on the dataflow programming model. Many of them rely on the concept of *single assignment* variables, i.e. variables which can only be assigned once. This concept avoids the Von Neumann model memory problems and a program is better suited for translation to DFGs.

Some of the popular early dataflow programming languages are :

- Textual Dataflow Language (TDFL) [35], developed in 1975, is considered to be the first dataflow language. A program in TDFL consist of series of modules (called procedures in some languages). Each module then consists of statements that can be assignments, conditional statements or calls to another module. Iterations are not directly supported but modules can call themselves iteratively.

- LAU [45] was developed in 1976 for the LAU static dataflow architecture by the computer structure group of Onera-Cert in France. It was a single assignment language. It provided explicit parallelism through the *expand* keyword.

- Lucid [46], the best known of all dataflow languages, was not originally developed as a dataflow language, but as a functional language to enable formal proofs. The objective was to write real-life program in a purely declarative style to enable verification. But later Lucid's functional and single assignment semantics established it to be a dataflow language [47].

- Id [48] was developed to write operating systems but without sequential controls and memory cells. Thus the language had single assignment and was block structured and expression based.

- There are several 'Manchester Languages' including DCBL [49],SISAL [50] and LAPSE [51] developed for the Manchester dataflow machine [52].

2.3.3 Dataflow Programming Languages for FPGAs

In this section, we will discuss some of the dataflow programming languages closely related to our work, namely include CAL, Canals, StreamIT and FPGA Brook. The CAPH language, which also belongs to this category, and on which our work is based, will be described separately in chapter 3.

2.3.3.1 CAL (Caltrop Actor Language)

CAL [53, 54], a dataflow/actor-oriented language is based on the Actor model of computation [55] for dataflow systems. The basic concepts of CAL have a natural resemblance to these systems. A dataflow model is described in CAL by a set of independent `actors` and their connections (called a `network` of actors).

An `actor` has a set of input and outputs which are used to communicate with other actors by exchanging data tokens. State variables are used to keep track of the internal state of an actor. The behavior of an actor is described using a set of actions. There must be at least one action in an actor. In the case of more than one action, CAL provides scheduling concepts to control the execution order of actions. The execution of an action depends on its internal state and the values available at input. During execution, an action can do all or any one of the following : change the state variable(s), read values from input and write values at output. It is also important to note that action execution is an atomic operation. At a time, only one action will be in execution.

A simple example of a CAL `increment` actor is given in listing 2.1, it has one input port `t` and one output port `s`, all of type integer [56]. This actor contains one action that consumes

one token from input port, and produces one token on the output port. This action will execute when data token is available at input port.

Listing 2.1: A Simple increment actor in CAL

```
actor Inc() integer t => integer s :
    action [a] => [inc]
        do
        inc := a + 1;
    end
end
```

In the above example, it is also possible to use a type variable `T` to create generic a type actor. The advantage is to declare actors with different types from one generic actor instead of writing a separate actor for each type.

In order to implement an application, CAL actors are connected to each other to form a `network` of actors. This is done by connecting the input and output ports of actors with each other. The connections are made with the help of FIFOs. CAL does not provide any explicit scheduling between actors, which means that the resulting system is entirely self-scheduling based on the actual flow of tokens. The declaration of a network of actors consisting of two actors, `inc` and `sum`, as shown in Fig. 2.13 is given in listing 2.2.

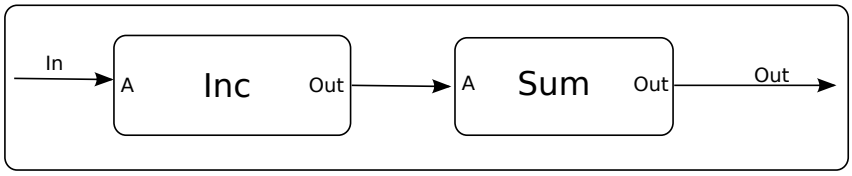


Figure 2.13: CAL dataflow network

Listing 2.2: CAL network declaration

```
network  Sum () In => Out:
    entities
        inc = Inc();
        sum = Sum();
    structure
        In --> inc.A;
        inc.Out --> sum.A;
        sum.Out -- > Out;
end
```

CAL takes care of low level communication details (e.g. message passing protocols) which helps designers to focus on `actors` and their connection to form a `network`. However, designers are provided control over connection communication parameters like length of FIFOs and the type of data exchanged.

When generating hardware implementations from networks of CAL actors, each actor is translated separately, and the resulting RTL descriptions are connected using FIFOs. Actors interact with FIFOs using a handshake protocol, which allows them to sense when a token is available or when a FIFO is full.

CAL has been selected by ISO/IEC for the definition of new MPEG standard called Reconfigurable Video Coding (RVC) [57]. All tools related to CAL are available under Orcc (Open RVC-CAL Compiler) [58] which is an update of the previous set of tools available under the Open Dataflow environment (OpenDF for short) [56]. It contains back ends for the generation of HDL [59], C [60] and Java [61]. MPEG Reconfigurable Video Coding framework has been implemented using CAL [62, 63]. CAL has many similarities with the CAPH language used in this thesis. Some elements of comparison are given in section 6.4.

2.3.3.2 Canals

Canals [64, 65], another dataflow language is based on *nodes* and *links*. The former consists of `kernels` and `networks` and latter is `channel` used to connected the nodes. `Kernels` are the basic computing unit of the Canals language. A kernel performs computation on input data and results are written on output. A `kernel` consist of three sections : an obligatory `work` block, a section for variable declarations and a section to initialize specific operations. Actual computations are performed inside a `work` block by using the *Canals Kernel Language*. This is a sequential language having syntax similar to many programming languages. All variables declared inside a `kernel` are local and cannot be accessed from outside. The values of these variables are changed during the execution of a `kernel`.

A new `kernel` is defined by using keyword `kernel` along with unique name and data types of input and output. The number of inputs read and the number of outputs written during the execution of `kernel` are specified by `get` and `put` keywords in the header of the `work` block. For example, `work get 1 put 1` in the header means that during the execution the `kernel` will read one value from input and write one value at the output. The code for a simple kernel declaration in Canals named **inc**, with input data type **dt1** and output data type **dt2** and consuming one element from input and producing one element at output during execution, is given in listing 2.3.

Listing 2.3: A Simple increment kernel declaration in Canals

```
kernel dt1 -> dt2 inc
{
   variable dt1 myIn;
   variable dt2 myOut;
        work get 1 put 1 {
        myIn = get();
        myOut = myIn + 1 ;
        put(myOut);
        }
}
```

The data types `dt1` and `dt2` are specified using the keyword `datadef`.

`Channels` are memory buffers used to store data between connected `kernels`. A channel is specified using the `channel` keyword along with a name and a data type. The capacity of the channel is described in the body. Canals also provides one pre-defined channel type called a `generic-channel`. It is an unbounded FIFO queue for any defined data type. Furthermore, in order to distribute and collect data, channels support `scatter` and `gather` operations. The former is used to distribute data from one input channel to many output channels and the latter is used to collect data from many channels.

In order to form a working application, a `network` is defined using the `network` keyword. A network is defined by giving it a name and data types of input and output. Elements are added to a network by using the statements `add-network`, `add-channel`, `add-kernel`, `add-scatter` and `add-gather`. After adding these elements, they are connected by `connect` statements. The connect statements must form at least one valid data flow path between an incoming data port and an outgoing data port which are donated by `NETWORK-IN` and `NETWORK-OUT` respectively. Other networks can also be added into a network. Although there can be many networks, there is always one top-level network defined as `network void -> void`. This top-level network acts as starting point for Canals program. An example of a network shown in Fig. 2.14 is defined in Canals as given in listing 2.4.

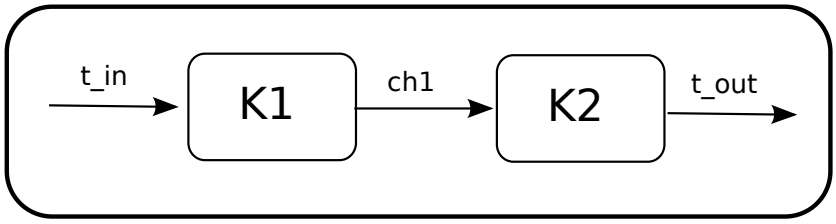


Figure 2.14: Canals dataflow network

Listing 2.4: Network declaration in Canals

```
network t_in -> t_out N
{
    add_channel ch1 <generic_channel>;

    add_kernel K1 <inc>;
    add_kernel K2 <sum>;

    connect NETWORK_IN -> K1 -> ch1 -> K2 -> NETWORK_OUT;
}
```

The Canals compiler first generates a behavioral model, a mapping model and an architecture model from the input code. All three are then combined to form an implementation model which is platform independent. This model is used by different backends to generate code with the help of Hardware Abstraction Layer (HAL) which contains the communication mechanism for the target architecture.

The MPEG Reconfigurable Video Coding framework has been implemented in Canals [64]. In [65], the JPEG encoder was implemented on an Altera FPGA using the Canals back end for FPGA.

2.3.3.3 StreamIT

StreamIT [66, 67] is an architecture-independent programming language for implementing high-performance streaming applications, by introducing stream-specific abstractions. This basic computational unit is called a `filter`. To build an application filters are connected through `streams`. StreamIt is based on the synchronous data flow (SDF) MoC but differs from this model by introducing multiple execution steps for filters, an option for declaring the

input/output rate of filter to be dynamic, teleport messaging[2], peeking (i.e. reading elements from input queue without deleting) and allowing filters to input and output a number of elements during initialization.

A `filter` consists of a single input channel and single output channel. Each `filter` is completely independent of the others and all communications between filters take place through input and output channels. It consists of two stages of execution: `initialization` and `steady state`. During initialization, the parameters to a filter are resolved to constants and the `init` function is called. During steady state execution, the `work` function is called repeatedly. It is also possible to write a `prework` function which is called once between `init` and `work` [68]. The work function repeatedly executes as soon as sufficient data is available on its input FIFO (queue). It reads data from its input queues using `pop` operations, writes data to its output queue using `push` operations and can also inspect inputs without removing them from the FIFO using a peek operation. The number of elements to push, pop or peek is declared in the declaration section of the work function. The example of an increment filter in StreamIT is given in listing 2.5.

Listing 2.5: A simple increment filter in StreamIT

```
int->int filter inc () {
   int result;
   init {
        result = 0;
        }
   }
   work push 1 pop 1  {
        result = pop()+1;
        push(result);
   }
}
```

In StreamIT an application is build by connecting filters into stream graphs. To accomplish this, three hierarchical stream primitives are provided : `pipeline`, `splitjoin`, and `feedbackloop` [69]. The pipeline structure creates a serial composition of streams by connecting inputs and outputs of filters to each others. A `splitjoin` specifies parallel streams that diverge from a common splitter and merge into a common joiner[3]. And a feedbackloop structure creates a cycle in the stream graph. The `add` keyword is used to instantiate and add a new filter to current stream graph. A simple stream graph as shown in Fig. 2.15, consisting of four filters in a pipeline can be written as in listing 2.6:

[2]To send a message from a filter's work function to change a parameter in another filter
[3]It is implemented with the help of data reordering primitives e.g.`duplicate`,`roundrobin`

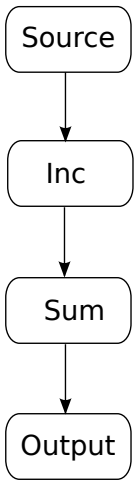


Figure 2.15: StreamIT dataflow network

Listing 2.6: Network declaration in StreamIT

```
int -> int pipeline Main() {
    add Source();
    add inc();
    add sum();
    add Output();
}
```

StreamIt was originally designed for the RAW[4] (Reconfigurable Architecture Workstation) machine [70], but more recently it has been used to introduce Optimus, an optimizing synthesis compiler for streaming applications on FPGAs [69]. It generates efficient Verilog HDL by performing many optimizations. These optimizations include Queue Allocation, Queue Access Fusion and Flip-Flop Elimination which effect space (area) and time (throughput) of the generated circuit. The first optimization reduces the size of the FIFO queues, the second fuses multiple queue operations into a single wider one and last identifies and eliminates redundant registers. The filters are synthesized using hardware templates and all templates are connected using FIFOs. The set of experiments implemented using this Optimus compiler includes : FFT (Fast Fourier Transform), parallel adder, bubble sort, merge sort, inverse DCT (Discrete Cosine Transform), DES (Data Encryption Standard) and matrix multiply [69].

2.3.3.4 FPGA Brook

FPGA Brook is a streaming programming language based on the programming languages Brook [71] and GPU brook [72]. The former is used to target multiprocessors [73] and the latter is a variant specifically designed for GPUs [74]. In [75], Brook is used to target applications on FPGAs by using the open source GPU brook compiler [72]. This version, called FPGA Brook, extends the Brook syntax to include streams and kernels. Computations are performed by `kernels` on input `streams`. Streams are collections of the data same as arrays but the elements are mutually independent. Streams are declared using characters < and > instead of square brackets. A kernel can exploit data-level parallelism by operating on individual stream

[4] A tiled multicore architecture

elements as all are independent of each other. Data-level parallelism can also be achieved by instantiating different instances of a kernel, each working on a part of the input stream. This is called kernel `replication` and is implemented using the `speedup` pragma statement. A special `reduction` kernel uses several elements of the input stream to produce one element of the output stream. There are also many stream operators for transforming input streams. For example, the `StreamReapeat` operator creates an output stream by repeating elements of the input stream. A `stencil` operator selects several elements of the input stream to create the output stream. `StreamRead` and `StreamWrite` operators are used to read (resp. write) input (resp. output) to/from memory. Although, Brook is based on the C programming language, it limits the usage of many features of the C language to make it easier for the compiler to analyze programs and extract parallelism [76]. The example of a simple increment filter in FPGA Brook is given in listing 2.7. The network depicted in Fig. 2.16 can be defined in FPGA Brook as given in listing 2.8.

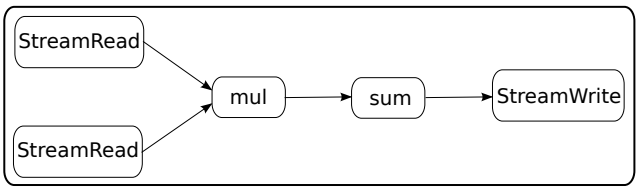


Figure 2.16: FPGA Brook dataflow network

Listing 2.7: A Simple increment kernel in FPGA brook

```
kernel void inc (int a<>, int c<>)
     {
      c = a+1;
     }
```

Listing 2.8: Network declaration in FPGA brook

```
void main ()
     {
      int Astr<1,N>, Tstr<1,N>, ystr<1,N>;
      int A[1][N], R[1][N];

      streamRead (Astr, A);
      inc (Astr, Istr);
      sum (Istr, Rstr);
      streamWrite (Rstr, R);
     }
```

FPGA Brook does not directly generate HDL code for FPGA implementation. Instead the design flow consist of two steps. In the first step, the program written in FPGA Brook is converted to C code using C2H directives. C2H [77] is Altera's high level synthesis tool which converts C-like code to HDL. The first step converts kernels to C2H functions to be implemented as hardware accelerators and also generates SOPC system description. In the second step, Verilog HDL is generated from this code using C2H. The Verilog code is then synthesized into FPGA logic by the Quartus II CAD tool [78] to produce the FPGA programming file, which can then be used to program the FPGA device.

Applications implemented in FPGA Brook include FIR Filter, Two-Dimensional Convolution, Zigzag Ordering in MPEG-2, Inverse Quantization in MPEG-2, Saturation in MPEG-2 and Mismatch Control in MPEG-2 [76].

2.3.4 Conclusion

Table2.1 summarizes the difference between the languages described in the previous section. Two of the languages (Canals and FPGA Brook) can only generate HDL code for a specific target, so they cannot be called "generic" programming languages for FPGAs. Out of the other two languages, CAL code has to undergo some changes to be executed on Altera FPGA. Furthermore, all these languages rely on some textual language or graphical tool to describe networks which is often a complicated task for big applications.

CAPH is introduced in the next chapter with the objective to overcome these limitations. The following characteristics of CAPH differentiate it from other related languages described in the previous section.

CAPH generates efficient VHDL code as compared to other languages (results of the comparison with one language (CAL) will be presented in chapter 6). But this efficiency comes at the cost of expressivity at the language level. The CAPH languages offers less features compared to CAL. So this efficiency is the result of a trade off with expressivity.

CAPH is based on formal semantics, which describes by a mathematical model all the possible computations performed by the language. The advantages of using this approach for CAPH are two fold. First, the transformation from high-level CAPH code to hardware-level VHDL code is described formally. It helps generate accurate VHDL code based on mathematical modeling of formal semantics. Second, the reference interpreter of the language is developed in a systematic way which is used to evaluate the accuracy of results generated by backend code.

In CAPH, it is possible to describe arbitrarily complex data structures as well as actor descriptions operating on these values. This means it can describe and operate on non-regular and/or variable-size data structure. The objective is to target a large domain of applications instead of restricting them to signal or image processing. This is achieved by separating the tokens into two categories: *data tokens*(having actual values) and *control tokens*(used as structuring delimiters).

The CAPH network sub-language is a small functional language which makes it easier to describe complex dataflow graphs using a set of functional equations. It offers a better way to describe application as compared to explicitly describing actor connections, even with the help of graphical interface. This will be further illustrated with the help of complex examples in next chapter.

	CAL	Canals	StreamIt	FPGA Brook	CAPH
Computing Unit	Actor	Kernel	Filter	Kernel	Actor
Dataflow MoC	Dynamic Dataflow	Synchronous Dataflow	Synchronous Dataflow	Static Dataflow	Dynamic Dataflow
Output	Verilog HDL	Code for Altera NIOS II processor	Verilog HDL	C Code with C2H directives	VHDL
Distribute data	No	Yes	Yes	Yes	Yes
Gather data	No	Yes	Yes	Yes	No
Affiliation	IETR, France and EPFL,Switzerland	Abo Akademi Univ., Finalnd	MIT, USA	Univ. of Toronto, Canada	Institut Pascal, France

Table 2.1: Dataflow Languages for FPGAs

Chapter 3

The CAPH language

The description of programming languages/environments for FPGAs in the previous chapter suggests that there is a still need to explore this area to use FPGAs to their full potential. Recently, considerable attention has been given to dataflow/actor-oriented model, as shown in section 2.3.3 of the last chapter. One idea introduced by the research community is to restrict the domain of applications by designing a domain specific language (DSL) to better utilize FPGAs for that specific domain. The CAPH [79] language is an example of this approach. CAPH is a high-level language for implementing stream-processing applications on FPGAs relying the dataflow/actor-oriented model of computation (MoC). It mainly differs from the languages introduced in section 2.3.3 by the constituents used to describe actors and the network representation respectively and also by introducing a way to represent streams containing arbitrarily structured data.

The objective of this chapter is to give a brief introduction to the CAPH language. Here CAPH v 1.6 is described, some changes have occurred since in the latest version. This chapter will describe its main features and how an application can be described. The complete language definition, describing concrete and abstract syntaxes and formal semantics, is given in the *Language Reference Manual(LRM)* [3].

The CAPH compiler generates SystemC and synthesizable VHDL code. The chapter starts by first describing the types supported by CAPH. Section 3.2 describes the different parts of a typical CAPH program which are combined to form a complete application. Section 3.3 gives an insight of the tools and design flow supported by the CAPH compiler.

3.1 CAPH Types

3.1.1 Base Types

The CAPH type system is polymorphic. Base types include signed and unsigned fixed-precision integers and booleans.

A signed integer `a` of size 8 bits is declared by the following syntax :

```
var a : signed<8>
```

The range of the above declared integer is -2^7 to $2^7 - 1$.

Similarly, a unsigned integers `c` of size 8 bits is declared as :

```
var c : unsigned<8>
```

The range of the above declared variable is 0 to $2^8 - 1$.

A boolean value is declared as :

```
var e : bool
```

Floating-point values are not supported, since they are not directly supported by VHDL synthesis tools. Built in operators on scalar types are given in Table 3.1. All of these operators are supported by simulator and are automatically translated to SystemC and VHDL implementation.

3.1.2 Structured Types

Two structured types are supported by CAPH : *arrays* and *data/control discriminated values (dc)*.

Table 3.1: Builtin operators on scalar types

bool	&& || !
unsigned **signed**	+ - * / % < <= = >= > land lor lnand lnor lxor lxnor lnot (bitwise operations) << >> (logical shift)

3.1.2.1 Arrays

Arrays are 1D or 2D collection of scalar type. Arrays have a fixed type and size, which are defined at declaration. For an array of size N, indexes range from 0 to N-1. An array can be initialized at declaration. To update an array, a new array is built from the previous one by modifying some of the elements in the old array.

A 1D array is declared by the syntax :

```
var name : type array[size]
```

Where `var` and `array` are keywords.

An array `a` containing 5 values of type `unsigned<8>` is declared as:

```
var a : unsigned<8> array[5]
```

A 2D array is declared as:

```
var name : type array[size,size]
```

It is also possible to initialize elements of an array. The following declaration initializes all the elements of array with zero :

```
var c : unsigned<8> array[10] = [ 0 : 10 ]
```

Whereas the following declarations initializes all elements with a different value :

```
var d : unsigned<8> array[15] = [i in 0..14 <- i]
var e : unsigned<8> array[5] = [6,5,91,2,30]
```

Similarly a 2D array is initialized as:

```
var f : unsigned<8> array[3,3] = [[1,2,3],[4,5,6],[8,9,10]]
```

To update an element, the following syntax is used :

```
name := name[index<-value]
```

For example, to update third element of array `a` with value 8, following code is used :

```
a := a[3<-8]
```

Some examples describing the use of arrays in CAPH are given in section 3.2.4.1.

3.1.2.2 DC (Data/Control) Type

The **dc** (data/control) type is used to represent streams containing arbitrarily structured data. This structuring is achieved by dividing tokens circulating on channels and manipulated by actors into two categories : data tokens (carrying values) and control tokens (acting as structuring delimiters). Only two types of control tokens are used to achieve this : one for the

start of the structure, represented by `'<'` and other for end of the structure, represented by `'>'`. For example, an image can be described as a list of lines, as depicted in Fig. 3.1, whereas the stream

```
<<<41 120> 44><<12 73> 58><<52 211> 7>>
```

may represent, for example, a list of points of interest, each inner pair consisting of its coordinates along with an attribute value.

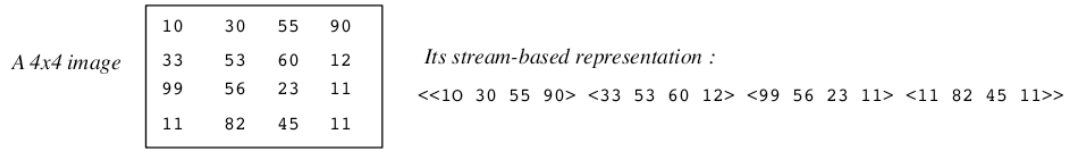


Figure 3.1: The structured stream representation of a 4x4 image

3.2 Program Structure

A CAPH program consists of a set of declarations. These declarations can be divided into five categories :

- Type declarations
- Global declarations
- Actor declarations
- I/O declarations
- Network declarations

3.2.1 Type Declarations

Type declarations are currently restricted to type synonyms: they give a name to an already existing type.

Example:

```
type byte = unsigned<8>
type pixel = signed<8> dc;
```

3.2.2 Global Declarations

Global declarations consist of constants and functions.

Constant declarations are used to assign value to a constant.

Example:

```
const scale = 20;
const kernel = [1,3,1];
```

Function declarations are used to map identifier(s) to an expression. The type signature can be optionally provided to give the type of the function (otherwise inferred by the compiler[1]).

Example:

```
function abs x = if x < 0 then 0-x else x;
function dec x = x-1 : signed<8> -> signed<8>;
```

External function declarations make it possible to use already written functions in SystemC or VHDL. The type of the "imported" function must be provided.

Example:

```
function abs x = extern "abs_c","abs_vhld","abs_ml" : unsigned<16>->unsigned<16>;
```

The above example uses three implementations of the function abs : `abs_c`, `abs_vhld`, `abs_ml` in SystemC, VHDL and Caml respectively. The Caml implementation of the function is needed for program simulation. The files containing these functions are to be kept in the same directory when compiling the respective code. Caml function also needs to be registered using a dedicated function. Programmer should be careful about the actual type signature of the function and the one provided in the declaration because the current version of compiler does not support type-based translation for foreign values.

3.2.3 I/O Declarations

I/O declarations are used to define the way the application interacts with the operating system (simulation) or the physical devices (VHDL code on FPGA). This is done through `stream` declarations. A `stream` declaration includes a name, type, direction (input or output) and a device. The CAPH application will read data from the input device, process it and write results on the output device. When using the simulator, I/O devices will be files.

Example:

```
stream input : signed<8> from "camera";
stream output : signed<8> to "monitor";
```

3.2.4 Actor Declarations

In CAPH an actor declaration consists of an *interface* and a *body*.

The *interface part* consists of the name of the actor, an optional list of parameter(s) and a list of input(s) and output(s). Inputs, outputs and parameter(s) are all typed. The interface is the only part of an actor which is visible in the network declaration section, where inputs and outputs will be used to connect channels and parameters are given values.

[1]The type inferred by the compiler is sometimes too general : for example, the type inferred for the function `dec x = x-1` is `signed<`α`> -> signed<`α`>`, where α is a size variable.

The following example gives the interface of a very simple actor, having one input and one output and no parameters.

Example 1:

```
actor a1
   in  (a: unsigned<8>)
   out (b: unsigned<8>)
--next comes the body of the actor
```

The next example also includes a parameter in the actor declaration. It has two inputs and one output.

Example 2:

```
actor a2 (k: unsigned<4>)
   in  (a: signed<8>, b: signed<8>)
   out (b: signed<8>)
--next comes the body of the actor
```

The **body** of an actor is used to define its behavior. It consists of a set of optional *local variables* and a set of *transition rules*. Each variable declaration consists of a name, a type and an optional initial value. The scope of variables is limited to the actor they are declared and they keep their values during the successive executions of the actor. An actor variable can have an *enumerated* type. This type is only supported for actor variables.

```
var state : {S0,S1,S2}
```

The above declaration introduced a new type *state*. But this type and the corresponding data constructors (S0,S1,S2) are limited to the current actor.

The behavior of an actor is specified using a set of **transition rules**. Each transition rule is made of a *pattern* and an *expression*. A pattern involves input(s) and/or local variables. The former is used to inspect input tokens and the latter to inspect variable values. Similarly, an expression involves output(s) and/or local variables. The former are used to write tokens and the latter to update variables. At each activation, a *fireable* rule is searched for. The selection of the transition rule to be fired is decided by sequential pattern matching. A rule is said to be fireable if input(s) and variable(s) match the rule pattern and results can be produced on the outputs involved in the expression. If no fireable rule is found then the actor waits for the next activation.

Each rule is written as:

```
|  (pat1,...,patm) -> (exp1,...,expn)
```

Parenthesis can be omitted if there is only one pattern or expression. The "|" is optional for the first rule. The referred variable or input/output in a pattern or expression is defined in the *rule format*. It is declared after the keyword `rules` and before the transition rules as:

```
|  (id1,...,idm) -> (id'1,...,id'n)
```

Where *id* (resp. *id'*) designates input (resp. output) or variable and *id'* is output or variable. The pattern (resp. expression) "_" means ignore for input, i.e. don't read input (resp. don't write output) and don't care for local variables.

3.2.4.1 Examples

We now give a number of examples of actor declarations.

Example 1:

```
actor id ()
  in  (a : unsigned<8>)
  out (c : unsigned<8>)
rules a -> c
| v -> v
```

This is a simple identity actor with one input and one output, of type `unsigned<8>`, no parameter and no local variable. There is only one rule which states that if a token is available on input `a`, then the actor will read this token, bound it to value `v` and copy the same value `v` on output `c`.

Example 2:

```
actor add ()
  in  (a : unsigned<8>, b : unsigned<8>)
  out (c : unsigned<9>)
rules (a,b) -> c
| (x,y) -> x+y
```

This is an add actor. It consist of two inputs, one output and no parameter. The single rule states : the input tokens read from input `a` and `b` are bound to values `x` and `y` respectively, and the sum of these two values is written on the output `c`. So, for an input stream of `1,9,6,...` and `7,5,11,...` on `a` and `b` respectively, the output at `c` will be `8,14,17,...`

Example 3:

```
actor skip (k : unsigned<8>)
  in  (a : unsigned<8>)
  out (c : unsigned<8>)
rules a -> c
| v -> if v=k then _ else v
```

This example skips all input values which are equal to the value `k`, provided here as a parameter. The value of this parameter will be specified when the skip actor is instantiated at the network level. The rest of the input values are just sent at the output. So, for k=**2**, the input stream `1,2,5,9,2,6,8,9,2,...` will produce the output stream `1,5,9,6,8,9,...`.

Example 4:

```
actor sum ()
  in (a : signed<8> )
 out (c : signed<16> )
var s : signed<16> = 0
rules (a, s) -> (c,s)
| (v, s) -> (s, s+v)
```

The `sum` actor reads a sequence of values on its input and produces the integral of this sequence on its output. For example, if the input stream is `0,1,2,3,4,...`, the output stream will be `0,1,3,6,10,...`. A local variable `s`, with initial value 0, is used to keep track of the running sum. The only transition rule can be read as follows : when a token carrying a value `v` is available on input `a`, then read it, write the current value of variable `s` on output `c` and add `v` to `s`.

Example 5:

```
actor incdec ()
  in (a : signed<5>)
 out (c : signed<5>)
var op : {Inc,Dec} = Inc
rules ( op, a ) -> ( c, op)
| ( Inc, v ) -> (v+1, Dec)
| ( Dec, v ) -> (v-1, Inc)
```

The `incdec` actor alternately increments and decrements the input stream and sends the result to the output. For example, if the input stream is `8,4,6,2,...` , then the output stream will be `9,3,7,1,...`. It uses a local variable (`op`) to keep track of which operation to perform at the next activation. The type of this variable is a (locally defined) enumerated type. There are two transition rules. The first (resp. second) transition rule says : If `op` is `'Inc'` (resp. `'Dec'`) and a token carrying a value `v` is available on input `a` then consume the corresponding token, write a token carrying value v+1 (resp. v-1) to output `c` and set `s` to `'Dec'` (resp. `'Inc'`).

Example 6:

```
actor rep ()
  in  (a : unsigned<3>)
  out (c : unsigned<8>)
var b : signed<8> array[8]=[0,10,20,30,40,50,60,70]
var i : unsigned<3> = 0
rules (a,b) -> c
| (i,b) -> b[i]
```

The above example will read the input stream and the value stored in array **b** at the index given by the input value is sent at output. For the input stream `5,2,6,3,4,5,...`, the output will be `50,20,60,30, 40,50,...`.

Example 7:

```
actor dec ()
  in (a : unsigned<8> dc)
 out (c : unsigned<8> dc)
rules a  -> c
| SoS    -> SoS
```

```
| EoS      -> EoS
| Data v -> Data (v-1)
```

The above example introduces the type `dc` for dealing with structured a stream of data. The input and output types are `unsigned<8> dc`. For input stream `<1,2,3>`, the output will be `<0,1,2>`. As described earlier, only two types of tokens (data and control) are used to describe structured stream. Here the patterns appearing on the right hand side of the transition rules are used to differentiate between these two type of tokens. The rules can be read as; if the input token is a control token (SoS or EoS), write the same token on output, if the input token is a data token, read the value, decrement it by one and write the resulting value on output. The `SoS`, `EoS` and `Data` constructors may be abbreviated as `'<`, `'>` and `'` respectively. So, the above example can also be written as:

Example 7_ bis:

```
actor dec ()
  in (a : unsigned<8> dc)
 out (c : unsigned<8> dc)
rules a -> c
| '< -> '<
| '> -> '>
| 'v -> '(v-1)
```

Example 8:

```
actor suml ()
  in (a : unsigned<8> dc)
 out (c : unsigned<16> dc)
var st : {S0,S1} = S0
var s : unsigned<8>
rules (st, a, s) -> (st, c, s)
| (S0, '<, _) -> (S1, _, 0)
| (S1, 'p, s) -> (S1, _, s+p)
| (S1, '>, s) -> (S0, s, _)
```

This example is a generalized form of the sum actor described in example 4. It accepts a sequence of lists and computes the sum of each list. For example, given input stream `<1 2> <7 8 9 3> <1 9 6>...` it will produce the output stream `3, 27, 16,...`. The `<` and `>` control tokens are used to delineate lists. The first transition rule detects the start of a list and initializes the accumulator `s` to 0. The second rule adds a list element to the accumulator. The last rule writes the accumulator on the output `c`.

Example 9:

```
actor scale ()
  in  (a : unsigned<8> dc)
  out (c : unsigned<8> dc)
var b : unsigned<8> array[5]=[1,2,3,4,5]
var i : unsigned<4> = 0
rules (a , b, i) -> (c, i)
| ('<, b, i) -> ( '<, 0)
| ('>, b, i) -> ( '>, _)
| ('v, b, i) -> (v*b[i], i+1)
```

The `scale` actor receives an input sequence of lists of length `5` and scales each element of the list by multiplying it with a value stored in an array **b**. For example, for the input stream `<1,2,3,4,5> <6,7,8,9,10>,...`, the output will be `<1,4,9,16,25> <6,14,24,36,50>,...`. The index `i` is used to access elements of the array `b`. At the start of each list, it is initialized from zero (in the first rule). The third rule will multiply the input value with the corresponding element stored in array and also increment the array index by 1.

Example 10:

```
actor d1p ()
  in (a : unsigned<8> dc)
 out (c : unsigned<8> dc)
var s : {S0,S1,S2} = S0
var z : unsigned<8>
rules ( s, a, z) -> ( s, c, z)
| (S0, '<, _) -> (S1, '<, _)
| (S1, '>, _) -> (S0, '>, _)
| (S1, '<, _) -> (S2, '<, 0)
| (S2, 'p, z) -> (S2, 'z, p)
| (S2, '>, _) -> (S1, '>, _)
```

The example delays each line of an image – represented here as a list of lists – by one pixel. For example, if the input image is the shown in Fig. 3.1 then the output image will depicted as in Fig. 3.2.

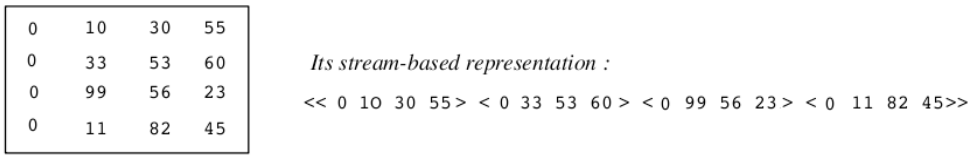


Figure 3.2: The image after application of a one-pixel delay per line

The CAPH description for the `d1p` actor uses two variables. Variable `z` keeps track of the previous pixel value, to be output when the current pixel is read and variable `st` acts as a state variable. The first and second rules handle the start and the end of an image (reading and writing a `<` and `>` control token respectively). The third rule is fired at each start of a new line; a `<` control token, indicating a start of line is produced and the variable `z` is set to 0. The fourth rule is fired for each pixel of a line; the previous pixel (stored in `z`) is output and the `z` variable updated. The last transition rule handles the end of a line.

3.2.5 Network Declarations

An application is described in a dataflow language as a dataflow graph (network) of actors. This is typically done by instantiating individual actors (creating nodes) and then connecting these nodes (a process called "wiring"). This can be done either *explicitly* (either textually or graphically) or *implicitly*. CAPH adopts the later approach, using a small purely functional and higher-order sub-language to describe network of actors. The basic idea is that dataflow graphs can be represented by functional equations. Here actors are functions, actor instantiation correspond to functional application and the connections between actors represent functional

dependencies. At the network language level, the only visible part of an actor is its interface (i.e. parameters and input(s)/output(s)). The network language instantiates these actors and wires them thus forming the dataflow graph describing the application. The input and output of the generated dataflow graph will be connected to external devices.

Example 1:

```
net o = dec i;
```

This simple example instantiates one actor having one input and one output, connecting its input to the application input `i` and its output to the application output `o`.

In the above case, the `net` keyword is used for binding network output. But it can also be used to create *wires* i.e. bind the output(s) of an actor to use it later, as exemplified below.

Example 2:

```
net (x,y) = dup i;
net o = add (shift x, y);
```

The above example describes the graph depicted in Fig. 3.3. Here, the `net` keyword serves to bind names to wires. The instantiation of the `dup` actor produces a node with two output wires, which are named `x` and `y` respectively. These wires are then used as inputs for the instances of the `shift` and `add` actors.

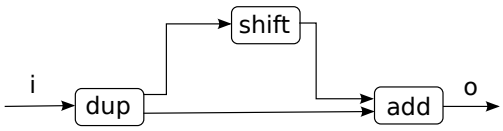


Figure 3.3: A dataflow network involving three actors

The CAPH approach of implicitly describing networks offers a significantly higher level of abstraction. In particular it saves the programmer from having to explicitly describe the wiring of channels between actors, a tedious and error-prone task, even with a help a graphical interface.

The CAPH approach for describing networks can be extended to define *higher-order wiring functions*. Higher-order wiring functions (HOWFs) can be used to define reusable polymorphic graph patterns, thus easing the process of building large applications from smaller ones. The concept is illustrated in Fig. 3.4. Here, `pipeline` is such a HOWF, taking both an actor and a wire as argument and building a sub-graph by "chaining" three instances of the actor.

The network graph in Fig. 3.3, can also be described by the following example by using a "triangle" HOWF :

Example 3:

```
net triangle (a,b,c) x =
 let (x1,x2) = a x in
  c (b x1, x2);
```

```
net o = triangle (dup,shift,add) i;
```


```
net pipeline (f,x) = f(f(f x));
net o = pipeline (shift, i);
```

Figure 3.4: A higher-order wiring function in CAPH

Another example is given in Fig. 3.5 where the triangle HOWF is used at three distinct levels of nesting. In the inner most call, its arguments are the three actors `dup`, `scale` and `add`. In the second inner call, the second argument is the *triangle* function which will result in a triangle pattern inside a triangle. In the outer call, again its second argument is the *triangle* function, which means the outer most triangle contains a triangle which in turn contains a triangle pattern as shown in Fig. 3.5.

```
net triangle (a,b,c) x =
 let (x1,x2) = a x in
  c (b x1, x2);

net o = triangle(
          dup,
          triangle(
            dup,
            triangle(dup,scale,add),
            add),
          add) i;
```

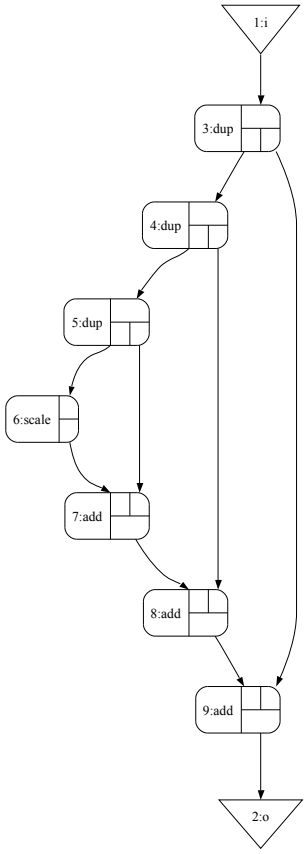


Figure 3.5: Building complex graph patterns using higher-order wiring functions

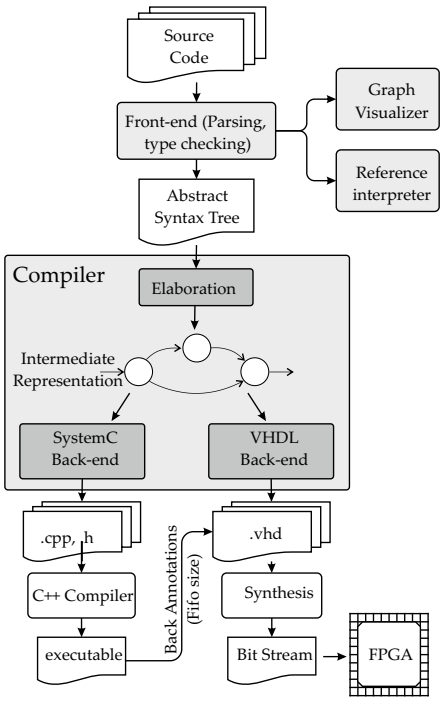


Figure 3.6: CAPH Toolset

3.3 Tools and design flow

The current tool chain supporting the CAPH language is shown in Fig. 3.6. It consists of a graph visualizer, a reference interpreter and a compiler producing SystemC and VHDL code.

3.3.1 Graph Visualizer

The **graph visualizer** is used for providing a graphical representation of a network of actors in `.dot` format [80] which can be visualized with the GRAPHVIZ suite of tools [81].

3.3.2 Reference Interpreter

The **reference interpreter** implements the dynamic semantics of the language, written in axiomatic style. It provides reference results to validate the correctness of SystemC and VHDL code before actual hardware implementation. It is also used to test and debug programs at early stage of development by reading/writing input/output to/from files. The input file can be text file containing data or an image file[2].It also provides several tracing and monitoring facilities. These include dumping:

- built in typing environment(for debug only)
- typed program (for debug only)
- print sized types using underlying representation
- static environment (for debug only)

[2]Routines are provided with the compiler to convert image files to textual format to be readable by reference interpreter

- intermediate representation (just before backends, for debug only)
- dynamic environment (for debug only)
- statistics about fifo usage after run
- fifo contents during run
- fifo usage in .vcd file during run

3.3.3 Compiler

The **compiler** is the core in this tool chain. It is comprised of the following parts:

3.3.3.1 Front-End

The **front-end** generates an Abstract Syntax Tree (AST) after parsing and type-checking. This generated AST is then used by the graph visualizer, the reference interpreter and elaboration.

3.3.3.2 Elaboration

The *elaboration phase* turns the Abstract Syntax Tree (AST) into a target-independent intermediate representation (IR). The IR is generated at two levels; one for the actor network and other for the description of each actor.

The network representation instantiates all the actors and connects them according to connections provided in *network declaration* part of CAPH program. At this level, actors are viewed as black boxes and the only visible parts of the actors are input(s), output(s) and parameter(s). This is achieved by using an *abstract interpretation* technique described in [82]. It evaluates the network declarations and for each representation of an actor, an instance of actor is created in the dataflow graph, then actors in the graph are connected. The connections between actors are represented by FIFO channels.

The IR for each actor is generated by converting the set of transition rules to a finite state machine (FSM). This FSM represents the actor rules in the form of target-independent state machine in which transitions are labeled with a condition/action set. The generation of the IR will be detailed in section 4.2.2.

3.3.3.3 Back-Ends

The current toolset consist of two backends : the first generates cycle-accurate SystemC code for simulation and profiling and second generates VHDL code for hardware synthesis. Before generating VHDL code, SystemC code is executed to provide some refinements in the VHDL implementation (for example, maximum depth needed for each FIFO is calculated, see section 4.3).

In the SystemC back end, each actor is converted to a SystemC module and these modules are connected through FIFO channels. The whole SystemC program is also converted to a module where input and output correspond to I/O streams. This program works as a *test bench* to test the whole design.

The VHDL back end is described in detail in the next chapter.

Chapter 4

The VHDL Backend

This chapter will describe in detail the steps followed for generating VHDL code from CAPH programs. The first part of the chapter will focus on data representation and how this is encoded at the VHDL level. The second part will describe the VHDL code generated by CAPH with the help of simple examples. The last part will discuss the important issue of FIFO implementation.

4.1 Data Representation

As described in section 3.1, an important feature of CAPH is its ability to describe *structured data streams*. For this, tokens are divided into two categories : *data tokens* and *control tokens*. This section describes how these tokens are encoded in VHDL, and how they are physically inserted in (resp. removed from) the input (resp. output) data stream.

4.1.1 Data/Control Encoding

As said in section 3.1.2.2, only two control tokens are required to encode arbitrary structured data streams : A "Start of Structure" (SoS) control token and a "End of Structure" (EoS) control token. These tokens are encoded using two extra bits in the binary representation of the stream values. These two bits are **00** for data token, **01** for the SoS control token and **10** for the EoS control token. The value **11** at these two bits indicates an error. Hence, for a stream carrying n-bit wide values

- Data tokens are encoded using n+2 bits as $\mathbf{00b}_{n-1}\mathbf{b}_{n-2}\mathbf{...b}_0$
- The SoS control tokens is encoded as $\mathbf{010}_{n-1}\mathbf{0}_{n-2}\mathbf{...0}_0$
- The EoS control tokens is encoded as $\mathbf{100}_{n-1}\mathbf{0}_{n-2}\mathbf{...0}_0$
- The configuration **11xx...x** is not used.

4.1.2 Token Insertion

A dedicated VHDL process is responsible for physically inserting control tokens into the input data steam. The implementation of this process depends on the structure of input data. It is implemented in VHDL with the help of a finite state machine. For example, for an input data stream consisting of a sequence of lists of equal length, the implementation of this process will be as follows : at the start, the `SoS` token is sent to the output. Next, an input value is read, the extra two bits containing **00** are added and sent to the output. This step will also keep a counter to track the number of data tokens sent : when this counter reaches the length of the list, it is reinitialized to zero and the `EoS` token is sent to the output. Then the process restarts. The graphical illustration of the process is given in figure 4.1. The delay of two clock cycles due to insertion of two control tokens is overcome because of the gap between the lists from the input device. The current CAPH toolset provides a set of parameterizable, predefined VHDL processes handling the automatic insertion of control tokens for the most commonly used data structures (lists and images). The VHDL process for adding control tokens in images will be described in section 6.1.

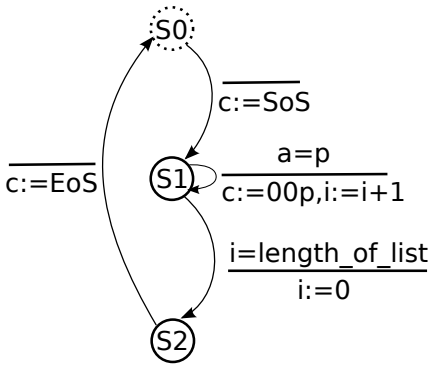


Figure 4.1: Finite state machine diagram for token insertion process

4.1.3 Token Removal

This process removes the extra bits from the output stream. It also adds a signal indicating whether the current data is valid or not. In case of control tokens, this bit will be set to **0** to indicate that data is invalid. In case of data token, this bit is set to **1** and data is send to the output by removing the extra two bits, leaving the remaining bits unchanged.

Fig. 4.2 illustrates the token insertion and removal process. Tokens are added to the raw input data stream. This stream is processed by the CAPH application and control tokens are removed from the output data steam. In this example, the input data stream is supposed to be structured as a sequence of lists of length **3**. The CAPH application will read the input data stream and copy the same data at output without any modifications.

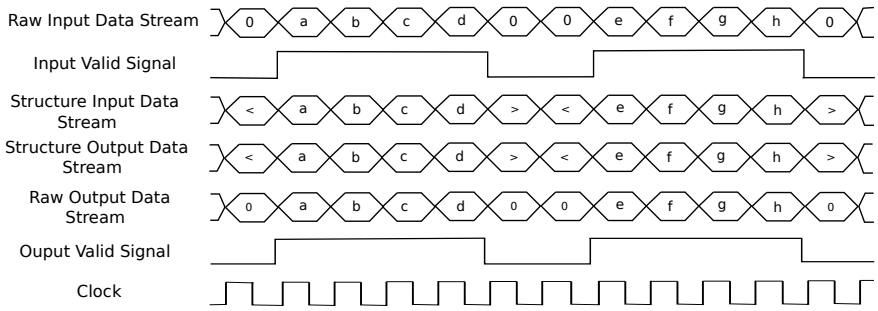


Figure 4.2: Token insertion and removal

4.2 VHDL Code Generation

In the sequel, we will discuss the process of VHDL code generation from a CAPH program using a simple example. This example calculates the horizontal derivative of an input stream consisting of a sequence of lines. The CAPH implementation involves two actors : `d1p` (delay one pixel) and `sub` (subtract) as shown in Fig. 4.3. The complete CAPH code for this example is given in listing 4.1.

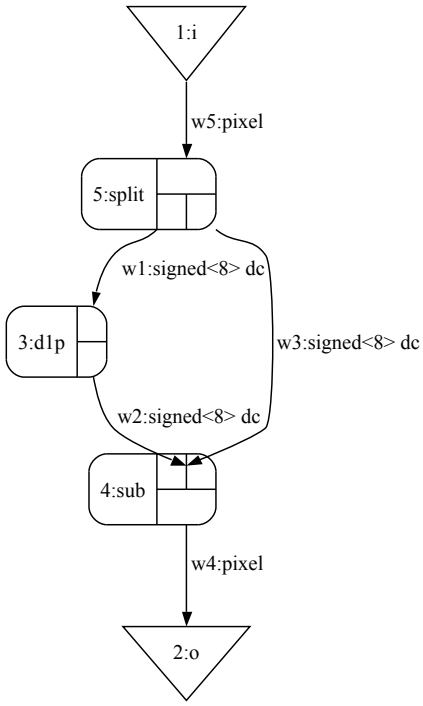


Figure 4.3: Graph of dx Example

Listing 4.1: CAPH application to calculate horizontal derivative

```
1  type byte = signed<8>;
2  type pixel = signed<8> dc;
3
4  actor d1p
5    in (a:pixel)
6    out (c:pixel)
7  var s : {S0,S1} = S0
8  var z : byte
9  rules (s, a, z) -> (s, c, z)
10 | (S0, '<, _) -> (S1, '<, 0)
11 | (S1, 'p, z) -> (S1, 'z, p)
12 | (S1, '>, _) -> (S0, '>, _);
13
14 actor sub
15   in (a:pixel, b:pixel)
16   out (c:pixel)
17 rules (a, b) -> c
18 | ('<, '<) -> '<
19 | ('v1, 'v2) -> '(v1-v2)
20 | ('>, '>) -> '>;
21
22 stream i:pixel from "sample.txt";
23 stream o:pixel to "result.txt";
24
25 net o = sub(i, d1p i);
```

The first part of the code (lines 1-2) consists of `type` declarations. Here `pixel` and `byte` types are declared.

The second part (lines 4-20) consists of the definition of two actors; `d1p` and `sub`.

The `d1p` actor (lines 4-12) accepts a stream of pixels structured as a (potentially infinite) sequence of lines of pixels –each line starting with the SoS (here denoted `'<`) control token and ending with the EoS (here denoted `'>`) control token– and delays each line by one pixel. For example, if the input stream is

< 1 2 3 4 > < 5 7 6 9 > ...

then the output stream produced by the **d1p** actor will be

< 0 1 2 3 > < 0 5 7 6 > ...

The description of this actor will be described in detail in section 5.2.

The `sub` actor (lines 14-20) has two inputs and one output. Depending on the input value, one of the three rules will be fired. In case of a pair of control tokens, the same token is written on output; in case of a pair of data tokens, the difference of two input values is written at output.

The next section (lines 21-22) defines I/O streams.

The last section (line 24) describes the dataflow network. It connects the input stream `i` to the `d1p` actor and to the first input of `sub` actor. The result of the `sub` actor is connected to the output stream `o`.

For example, if the input stream is

< 1 2 3 4 > < 5 7 6 9 > ...

then the output stream produced by this example will be

< 1 1 1 1 > < 5 2 -1 3 > ...

Generating the VHDL (as well as SystemC) code from this CAPH description involves two steps. The first is generating a dataflow graph from the network description and the second is generating the behavioral description of each actor. In the case of VHDL, the generated code consists of one network description file and one file for each actor. The next sections describe the code generated for the network file and for the `sub` actor (the code for the `d1p` actor will be described later in section 5.2).

4.2.1 VHDL code for the dataflow network

The network is depicted in Fig. 4.3. Note that the compiler has inserted a special node called `split`. The corresponding actor is responsible for *duplicating* the stream of tokens produced by the input `i`, in order to feed both the `d1p` and `sub` actors. The VHDL description for this network, produced by the compiler is given in listing 4.2. Lines 1-5 consist of `library` declarations. One is the standard `IEEE` library, the second the `dc` library containing the package operating on structured values and FIFOs used to connect actors.

Listing 4.2: VHDL code generated for dataflow network

```
1 library ieee;
2 library dc;
3 use ieee.std_logic_1164.all;
4 use dc.dcflow.all;
5 use ieee.std_logic_unsigned.all;
6
```

```
7  entity dx_net is
8    port (
9      w5_f: out std_logic;
10     w5: in std_logic_vector(9 downto 0);
11     w5_wr: in std_logic;
12     w13_e: out std_logic;
13     w13: out std_logic_vector(9 downto 0);
14     w13_rd: in std_logic;
15     clock: in std_logic;
16     reset: in std_logic
17     );
18 end dx_net;
19
20 architecture arch of dx_net is
21
22 component sub_act is
23    port (
24     a_empty: in std_logic;
25     a: in std_logic_vector(9 downto 0);
26     a_rd: out std_logic;
27     b_empty: in std_logic;
28     b: in std_logic_vector(9 downto 0);
29     b_rd: out std_logic;
30     c_full: in std_logic;
31     c: out std_logic_vector(9 downto 0);
32     c_wr: out std_logic;
33     clock: in std_logic;
34     reset: in std_logic
35     );
36 end component;
37
38
39 component d1p_act is
40    port (
41     a_empty: in std_logic;
42     a: in std_logic_vector(9 downto 0);
43     a_rd: out std_logic;
44     c_full: in std_logic;
45     c: out std_logic_vector(9 downto 0);
46     c_wr: out std_logic;
47     clock: in std_logic;
48     reset: in std_logic
49     );
50 end component;
51
52
53 signal w12_f : std_logic;
54 signal w12 : std_logic_vector(9 downto 0);
55 signal w12_wr : std_logic;
56 signal w10_f : std_logic;
57 signal w10 : std_logic_vector(9 downto 0);
58 signal w10_wr : std_logic;
59 signal w11_e : std_logic;
60 signal w11 : std_logic_vector(9 downto 0);
61 signal w11_rd : std_logic;
62 signal w8_f : std_logic;
63 signal w8 : std_logic_vector(9 downto 0);
```

```
64 signal w8_wr : std_logic;
65 signal w9_e : std_logic;
66 signal w9 : std_logic_vector(9 downto 0);
67 signal w9_rd : std_logic;
68 signal w6_f : std_logic;
69 signal w6 : std_logic_vector(9 downto 0);
70 signal w6_wr : std_logic;
71 signal w7_e : std_logic;
72 signal w7 : std_logic_vector(9 downto 0);
73 signal w7_rd : std_logic;
74
75 begin
76   F9: fifo_small generic map (4,10) port map(w12_f,w12,
77                              w12_wr,w13_e,w13,w13_rd,clock,reset);
78   F8: fifo_small generic map (4,10) port map(w10_f,w10,
79                              w10_wr,w11_e,w11,w11_rd,clock,reset);
80   F7: fifo_small generic map (4,10) port map(w8_f,w8,
81                                w8_wr,w9_e,w9,w9_rd,clock,reset);
82   F6: fifo_small generic map (4,10) port map(w6_f,w6,
83                                w6_wr,w7_e,w7,w7_rd,clock,reset);
84   S5: split2 generic map (10) port map(w5_f,w5,w5_wr,
85                                w10_f,w10,w10_wr,w6_f,w6,w6_wr);
86   B4: sub_act  port map(w7_e,w7,w7_rd,w9_e,w9,w9_rd,
87                                w12_f,w12,w12_wr,clock,reset);
88   B3: d1p_act  port map(w11_e,w11,w11_rd,w8_f,w8,w8_wr,clock,reset);
89 end arch;
```

Lines 7-18 declare the CAPH program as an entity. The input(s) and output(s) of this entity will be connected to the application I/Os. When simulating the program, these I/Os will generally be read/written from/to file(s). When the program will be implemented on an FPGA, these I/Os will generally be connected to the camera and display devices.

Lines 20-90 give the implementation of this entity in a *structural* way. First, a `component` is declared for each actor (two here lines 22-50). Then *signals* are declared which are used to connect FIFOs and actors. As shown in Fig. 4.4, each wire at the CAPH level is turned into a FIFO and six signals are used to connect the two actors.

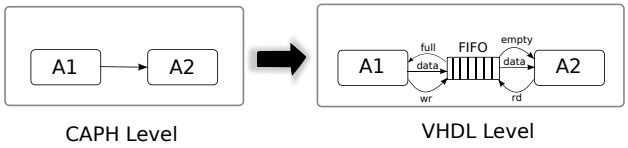


Figure 4.4: CAPH to VHDL transition of actor connectivity

The architecture of the FIFO is shown in Fig. 4.5, the interface of each FIFO component consists of eight signals : `clock`, `reset`, `full`, `input`, `write`, `empty`, `output` and `read`. These signals are used by actors to read/write to/from the FIFO. One FIFO will be used to connect at least two actors; one for writing and other for reading. The first three signals (`full`,`input`,`write`) are used by the writing actor. The `full` output signal tells whether the FIFO is full. If yes, the writing actor will wait until space is available for writing in the FIFO. The `input` signal corresponds to the data sent by the actor to be written on FIFO. The `write` signal is used by the writing actor to trigger the writing operation. The remaining three signals

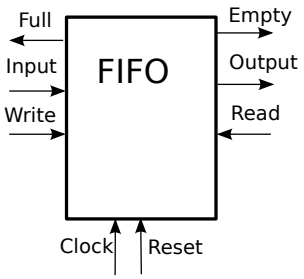


Figure 4.5: FIFO Architecture

(`empty`,`output` and `read`) are used by the reading actor. The `empty` output signal tells whether the FIFO is empty or not. If yes, the reading actor will wait until some data is available in the FIFO to be read. The `output` corresponds to the data sent by the FIFO. The `read` input signal is used by the actor to request a read operation. When this signal is asserted (and the FIFO is not empty), the FIFO updates its reading counter and writes the corresponding data on the `output` signal at the next rising edge of the clock.

Lines 86-88 instantiate each actor as a VHDL `component` and these components are used to form a network of actors through FIFO interconnections in lines 76-83. Since, in this case there are three actors (including the `split` actor inserted by the compiler), the complete network representation consists of four FIFOs and three actor components. The implementation of the `split` actor is provided in the `dc` library[1].

The correspondence between wires in Fig. 4.3 and the FIFOs in the above code is as follows : FIFO `F8` represents wire `w1`, FIFO `F6` represents wire `w3`, FIFO `F7` represents wire `w2` and FIFO `F9` represents wire `w4`. The two parameters provided to each FIFO component are its capacity and width in bits (see section 4.3).

The RTL view for the generated VHDL network file is shown in Fig. 4.6. The generated structure of the graph, consisting of actors connected through FIFOs can easily be recognized.

4.2.2 VHDL code for the sub actor

This section describes the VHDL code generated by CAPH for the `sub` actor. As described earlier, the VHDL code for each actor is generated in a separate file. The code for the `sub` actor is given in listing 4.3.

Listing 4.3: VHDL code generated for sub actor

```
1  library ieee;
2  library dc;
3  use ieee.std_logic_1164.all;
4  use dc.dcflow.all;
5  use ieee.std_logic_unsigned.all;
6
7  entity sub_act is
8     port (
9      a_empty: in std_logic;
10     a: in std_logic_vector(9 downto 0);
```

[1]This implementation contains different versions of this entity to support up to four duplications. This can further be increased to application's requirement.

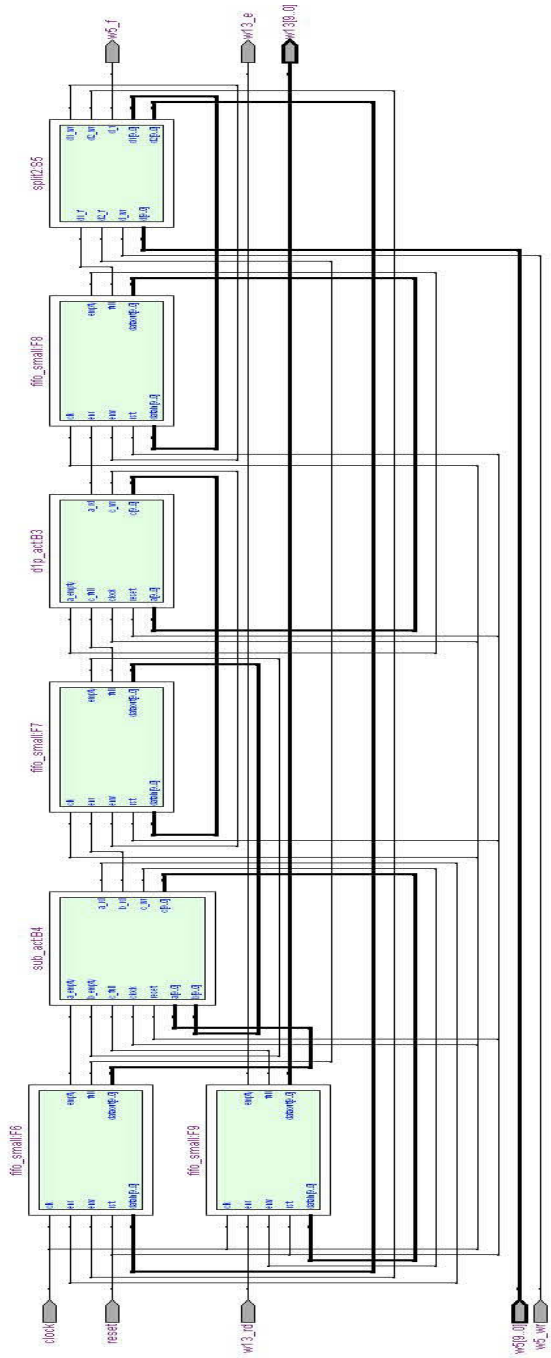


Figure 4.6: RTL diagram of network graph

```
11      a_rd: out std_logic;
12      b_empty: in std_logic;
13      b: in std_logic_vector(9 downto 0);
14      b_rd: out std_logic;
15      c_full: in std_logic;
16      c: out std_logic_vector(9 downto 0);
17      c_wr: out std_logic;
18      clock: in std_logic;
19      reset: in std_logic
20      );
21  end sub_act;
22
23  architecture FSM of sub_act is
24      type t_state is (R0,R1,R2,R3);
25      signal state: t_state;
26  begin
27    process(clock, reset)
28      variable v1_v : std_logic_vector(7 downto 0);
29      variable v2_v : std_logic_vector(7 downto 0);
30    begin
31      if (reset='0') then
32        state <= R0;
33        a_rd <= '0';
34        b_rd <= '0';
35        c_wr <= '0';
36      elsif rising_edge(clock) then
37        case state is
38          when R0 =>
39            if b_empty='0' and is_sos(b) and a_empty='0' and
40                is_sos(a) and c_full='0' then
41              b_rd <= '1';
42              a_rd <= '1';
43              c <= sos(10);
44              c_wr <= '1';
45              state <= R1;
46            elsif b_empty='0' and is_data(b) and a_empty='0'
47               and is_data(a) and c_full='0' then
48              v2_v := data_from(b);
49              b_rd <= '1';
50              v1_v := data_from(a);
51              a_rd <= '1';
52              c <= to_data(v1_v - v2_v,10,true);
53              c_wr <= '1';
54              state <= R2;
55            elsif b_empty='0' and is_eos(b) and a_empty='0'
56               and is_eos(a) and c_full='0' then
57              b_rd <= '1';
58              a_rd <= '1';
59              c <= eos(10);
60              c_wr <= '1';
61              state <= R3;
62           end if;
63          when R1 =>
64              b_rd <= '0';
65              a_rd <= '0';
66              c_wr <= '0';
67              state <= R0;
```

```
68          when R2 =>
69              b_rd <= '0';
70              a_rd <= '0';
71              c_wr <= '0';
72              state <= R0;
73          when R3 =>
74              b_rd <= '0';
75              a_rd <= '0';
76              c_wr <= '0';
77              state <= R0;
78        end case;
79      end if;
80    end process;
81 end FSM;
```

The first part (lines 1-5) consists of `library` declarations. One is the standard `IEEE` library, the second the `dc` library containing the package operating on structured values and FIFOs used to connect actors.

After library declarations, lines 7-21 declare the VHDL `entity` for the actor. Other than `clock` and `reset` signals, the remaining signals in the port declaration are for the input and output FIFOs. The signals starting with `a_` and `b_` are for the input FIFOs and `c_` for the output FIFO.

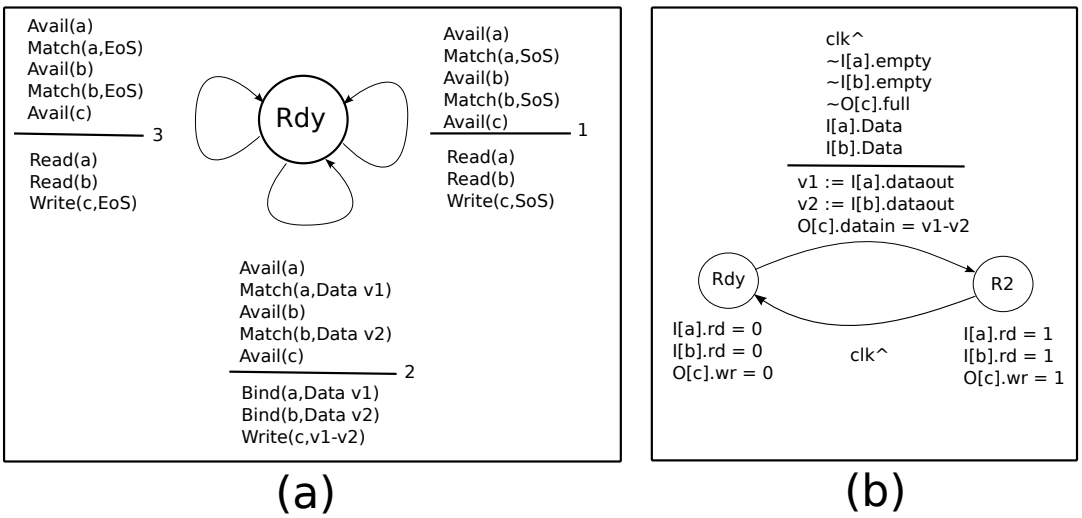


Figure 4.7: Intermediate Representation (IR) for the sub actor of listing 4.3 and transformation of the second rule.

In lines 23-81 the corresponding `architecture` is defined. The first two lines 24-25 declare a `state` signal. This signal is used to encode a finite state machine in VHDL. This FSM is derived from the rule-based specification of the actor. For this, the rules describing the behavior of the actor are first converted to an intermediate representation. This intermediate representation is then transformed into VHDL. The transformation process – described in detail in [79, 83]– is illustrated in Fig. 4.7-a using the sub actor as an example. In this figure , the small number besides each transition gives the number of the corresponding rule in the CAPH code. In Fig. 4.7-b, the transformation of one rule for VHDL is also given. For this transformation, FIFO signals are generated to control read (resp. write) operations on the input (resp. output) FIFO. The *Avail* condition on the input (resp. output) reflects the availability of input (resp.

output) for reading (resp. writing). The `rd` (resp. `wr`) signals perform the actual read (resp. write) operations. As these signals are asserted synchronously, an extra state is needed for each rule. This transformation described here is for the second rule in Fig. 4.7-b. The `clk^` represents the synchronizing clock signal, logical negation is denoted by the `~` prefix operation and `I[a]` (resp. `I[b]` and `O[c]`) represents the FIFO connected to input `a` (resp. input `b` and output `o`).

Lines 36-79 encode the resulting synchronous FSM. This FSM is initially in state `R0`. Then at each clock cycle it will perform some operations and possible move to another state depending on the availability of input(s), the value of input(s), the availability of room in the output FIFOs (to write the results) and the value of local variables. The availability and value of inputs are checked by calling the `is_sos`, `is_data` and `is_eos` functions, provided by `dc` library. In the case of data, input values are stored in variables `v1_v` and `v2_v` in lines 48 and 50 respectively. The difference between these two variables is calculated and the result is stored in the output FIFO by asserting the write signal for the output FIFO in lines 52 and 53. In case of control tokens (i.e. SoS/EoS), the same token is written on the output. In all three conditions, the FSM is moved to state `R1` (resp. `R2`,`R3`) in order to reset the `read` and `write` signals to zero. This means that when input (resp. output) FIFOs are available for reading (resp. writing), one rule will actually take two cycles to execute.

There is no local variable in the `sub` actor used to control execution of transition rules. When local variables are used (as in `d1p` actor), these variables are converted to signals in the corresponding VHDL process (see section 5.2).

The RTL view of generated VHDL for the the `sub` actor is shown in Fig. 4.8. The yellow box represents the *state machine* which is given in Fig. 4.9. The rest consists of *registers*, *equal* operators and *multiplexers*. The output from *equal* operators is used by the *multiplexers* to execute one of the if conditions in the `R0` state and results are stored in *registers* before being sent at output. The VHDL design for this actor, when compiled for an FPGA[2] utilizes 18 logic elements, none of the other resources (DSP, memory etc.) are used. The design operates at a maximum frequency of 390 MHz.

4.3 Dimensionning FIFOs

FIFOs play a key role in the CAPH implementation model. But when targeted for resource constrained devices like FPGAs, their usage must be strictly controlled. Therefore, considerable attention is given by the CAPH compiler to FIFO utilization.

4.3.1 FIFO size

First, the width (in bits) of each FIFO is deduced in a straightforward way from the type of the data it contains.

Assigning a depth (in places) to each FIFO is a more complex issue. If the capacity is too small, then deadlocks can occur at execution. If this capacity is too big, then some waste of resources occurs. Computing the optimal capacity at compile time is in general undecidable because the time required by an actor to produce its results can depend on the value of its inputs. The approach used by CAPH is to obtain an estimation of the required capacity using run-time

[2]using Quartus toolset for Stratix EP1S60 FPGA

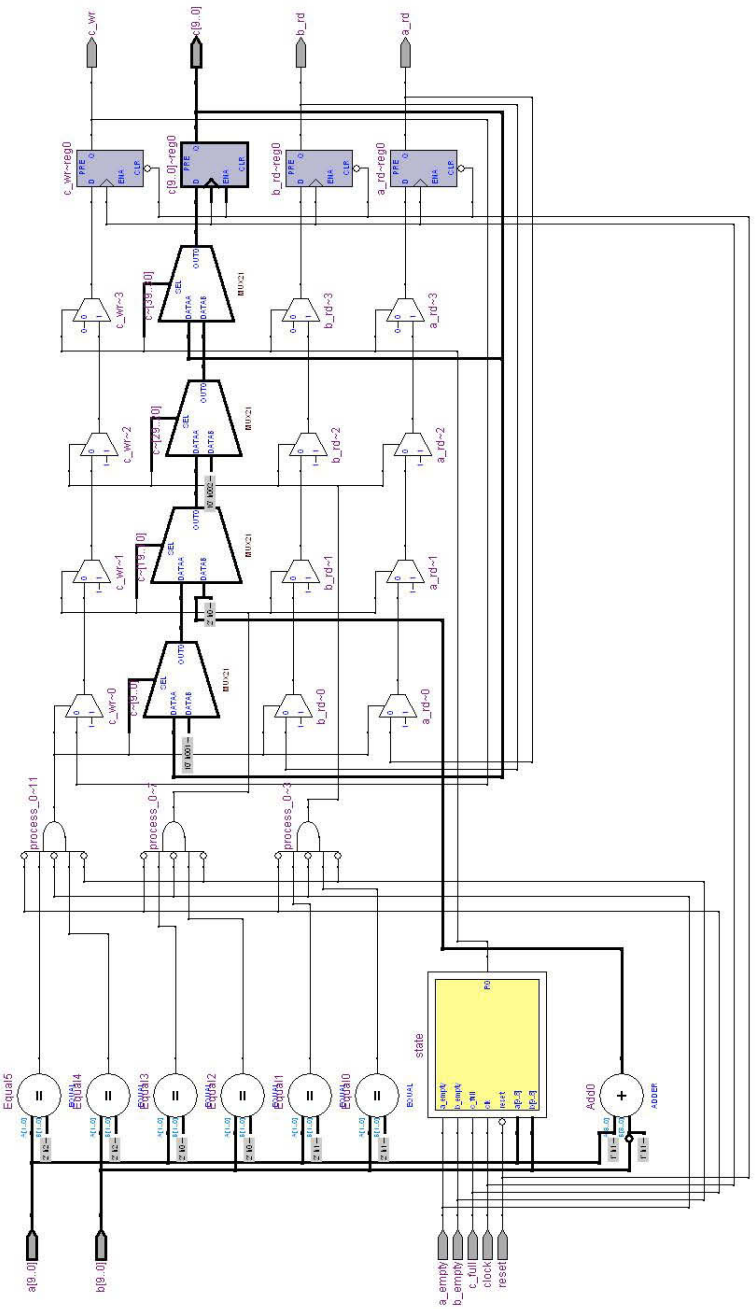


Figure 4.8: RTL diagram of sub actor

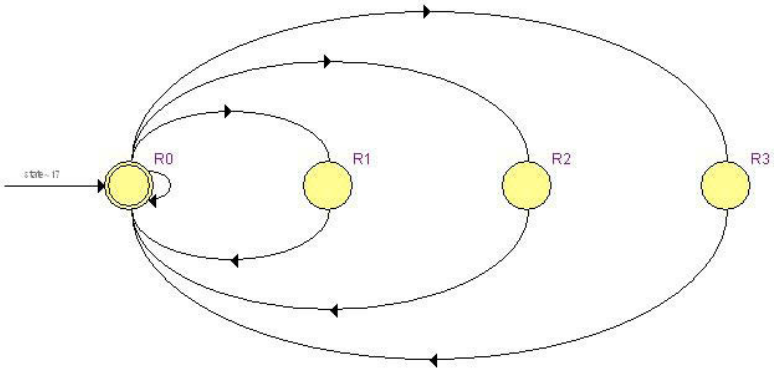


Figure 4.9: State machine generated by sub actor

profiling. This is the main role of the CAPH SystemC back end. For this, the code generated by the SystemC back end, when executed, monitors the run time occupancy of each FIFO and writes it to a specific annotation file. The content of this file for the example introduced in listing 4.1 and Fig. 4.3 is given in Fig. 4.10. Apart from the FIFO at wire `w3`, all other FIFOs have a maximum occupancy of one. The FIFO at `w3` has to store an extra value because the second input of `sub` actor (wire w2) will be available after a delay of one clock cycle because of the presence of the `d1p` actor in the corresponding path.

```
w5 fifo_max_occ = 1
w3 fifo_max_occ = 2
w2 fifo_max_occ = 1
w1 fifo_max_occ = 1
w4 fifo_max_occ = 1
```

Figure 4.10: Annotation generated by the SystemC code

The annotation file is used by the VHDL backend to assign accurate size to each FIFO. This explains the arrow labeled “back-annotations” in Fig. 3.6.

In the absence of annotation file, the VHDL backend assigns a default depth to each FIFO of the design. This default depth is 4. It can be changed with the compiler option *-vhdl_default_fifo_capacity*.

4.3.2 Actual FIFO Implementation

The actual implementation of the FIFO on the target device is also an important issue, impacting specifically the resource usage of the target FPGA. There are basically two ways to implement a FIFO : with registers or with embedded memory. The former is the simplest and fastest solution but quickly becomes inadvisable for large FIFOs because it consumes a large

number of logic elements.

In order to make optimal usage of FPGA resources, different options are provided by the compiler. The option *-vhdl_default_fifo_model* sets the FIFO model to use for a "small" FIFO which will be implemented with registers. The *-vhdl_big_fifo_model* option selects the model to use for "big" FIFO (which will be implemented with embedded memory). The *-vhdl_fifo_model_threshold* option is used to tell the compiler when to switch from the former model to the second : FIFOs having a capacity smaller than the specified threshold will be implemented with the "small" FIFO and those with capacity bigger than the threshold will be implemented with the "big" model.

Chapter 5

Examples

This chapter will highlight some features of CAPH using a set of selected examples. These features are highlighted either because they illustrate some relevant points from a programming perspective or raise interesting or challenging problems when translating the code to VHDL for FPGA.

5.1 Arithmetic

We first introduce the usage and translation of basic arithmetic and logic operators. The conversion of these operators from CAPH to VHDL boils down to selecting the right library operator or function.

Example 1 : add

This simple example reads two inputs and writes their sum on output. The CAPH code for this example is given in listing 5.1.

Listing 5.1: CAPH code for add example

```
actor add ()
  in  (a:signed<8>,b:signed<8>)
  out (c:signed<9>)
rules (a,b) -> c
| (a,b) -> a+b
;
```

The CAPH language uses the '+' operator for addition. The same operator is used in VHDL for addition. The part of the VHDL code representing the rule is :

```
if b_empty='0' and a_empty='0' and c_full='0' then
  b_v := b;
  b_rd <= '1';
  a_v := a;
  a_rd <= '1';
  c <= a_v + b_v;
  c_wr <= '1';
  state <= R1;
```

Here it can be seen that the same operator is used for addition in VHDL. Regarding type conversion, the inputs of type `signed<8>` in CAPH are converted to `std_logic_vector(7 downto 0)` in VHDL and one output of type `signed<9>` is converted to `std_logic_vector(8 downto 0)` in VHDL. The resulting conversion and size of vector will avoid any overflow during the operation.

Example 2 : shift

This example reads an input value, performs a left shift of two bits and writes the result on output. The CAPH code for this example is given in listing 5.2.

Listing 5.2: CAPH code for shift example

```
actor shift ()
  in  (a:signed<8>)
  out (c:signed<8>)
rules a -> c
| a -> a<<2
;
```

The VHDL code generated by the backend for the only rule is :

```
if a_empty='0' and c_full='0' then
  a_v := a;
  a_rd <= '1';
  c <= SHR(a_v,2);
  c_wr <= '1';
  state <= R1;
end if;
```

Here the '<<' CAPH operator is converted to the 'SHR' VHDL function.

The list of all the operators supported by CAPH is given in chapter 2 of the Language Reference Manual (LRM) [3].

Example 3 : Power of two series

The example reads an input value, and tells whether the input number is in the series of powers of two (1,2,4,8,...) or not. The CAPH code for this example is given in listing 5.3.

Listing 5.3: CAPH code for power of two series example

```
 in(a:unsigned<8>)
 out(c:bool)
rules a -> c
| v -> if(v land(v-1)=0) then true else false
;
```

After input and output declarations, the only rule reads input, does the logical and between the input value and one subtracted from the input and checks whether the result is equal to zero. If yes then it is a number from the above series otherwise it is not. The VHDL code generated for this rule is:

```
if a_empty='0' and c_full='0' then
    v_v := a;
    a_rd <= '1';
    if ( eq((v_v) AND ((v_v)-("00000001")),"00000000"))
        then c <= "1";
        else c <= "0";
    end if;
    c_wr <= '1';
    state <= R1;
end if;
```

Here the VHDL code generation is simple. A logical and operation is performed between input value and one subtracted from it and equality with zero is checked with function eq provided in the dc library. Depending on the result **1** or **0** is sent at output.

5.2 One-pixel delay

The corresponding actor has already been introduced in section 4.2. Its goal is to delay each input line by one pixel. For example, if the input is

$< a_1, a_2, \ldots, a_n >$

then the output will be

$< 0, a_1, a_2, \ldots, a_{n-1} >$.

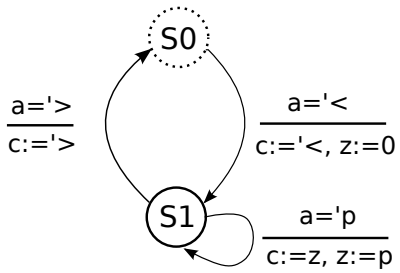


Figure 5.1: One-pixel delay actor state diagram

A functional description of this actor could be :

$d1p :< P_1, P_2, \ldots, P_n > \longrightarrow < 0, P_1, P_2, \ldots, P_{n-1} >$

The behavior of the actor is represented graphically in Fig. 5.1. In the figure, circles represent states (the dotted circle is initial state), the arrows are used for transition between states and text besides each arrow consists of action(s) and condition(s) (the text above the line is condition(s) and below the line is action(s)). The CAPH code for this example is given in listing 5.4.

Listing 5.4: CAPH code for one-pixel delay example

```
1  actor d1p ()
2    in (a:unsigned<8> dc)
3    out (c:unsigned<8> dc)
4  var s : {S0,S1} = S0
5  var z : unsigned<8>
6  rules (s, a, z) -> (s, c, z)
7  | (S0, '<, _) -> (S1, '<, 0)
8  | (S1, 'p, z) -> (S1, 'z, p)
9  | (S1, '>, _) -> (S0, '>, _)
10 ;
```

The type of the input and output is `unsigned<8> dc`, meaning that these I/Os are structured stream of `unsigned` 8 bit values. The *structuration* of the stream– by means of `'<` and `'>` control tokens– is essential here for expressing the behavior of the `d1p` actor in a generic[1] way.

Two local variables are used (declared in lines 4-5). Variable `s` keeps track of the actor "state". Its value is `s0` when the actor is idle, waiting for new line to process and `s1` when actor is processing a line. Variable `z` is used to memorize the previous pixel.

The behavior of actor is described with three rules. The first rule (line 8) handles the start of the line (reading and writing the start of line `<` control token). It initializes the variable `z` to zero and moves to state `s1` for processing the line. The second rule in line 9 handles the data tokens. It reads the input value and stores it in a temporary variable `p`. The value stored in variable `z` is written to the output, this variable is updated with the value read on input. The third rule in line 10 handles the end of the line (reading and writing the end of line `>` control token). It also moves back to state `s0` and starts waiting for the next line at input.

The VHDL file generated for this actor is given in listing 5.5.

[1] In a sense that it does not depend on the actual length of the input lines

Listing 5.5: VHDL code generated for d1p actor

```
1  library ieee;
2  library dc;
3  use ieee.std_logic_1164.all;
4  use dc.dcflow.all;
5  use ieee.std_logic_unsigned.all;
6
7  entity d1p_act is
8     port (
9      a_empty: in std_logic;
10     a: in std_logic_vector(9 downto 0);
11     a_rd: out std_logic;
12     c_full: in std_logic;
13     c: out std_logic_vector(9 downto 0);
14     c_wr: out std_logic;
15     clock: in std_logic;
16     reset: in std_logic
17     );
18 end d1p_act;
19
20 architecture FSM of d1p_act is
21     type t_state is (R0,R1,R2,R3);
22     type t_enum1 is (S0,S1,S2);
23     signal state: t_state;
24     signal s : t_enum1;
25     signal z : std_logic_vector(7 downto 0);
26 begin
27   actorprocess(clock, reset)
28     variable p_v : std_logic_vector(7 downto 0);
29     variable z_v : std_logic_vector(7 downto 0);
30   begin
31     if (reset='0') then
32       state <= R0;
33       s <= S0;
34       a_rd <= '0';
35       c_wr <= '0';
36     elsif rising_edge(clock) then
37       case state is
38         when R0 =>
39           if a_empty='0' and is_sos(a) and s=S0 and c_full='0' then
40             a_rd <= '1';
41             s <= S1;
42             c <= sos(10);
43             c_wr <= '1';
44             z <= "00000000";
45             state <= R1;
46           elsif a_empty='0' and is_data(a) and s=S1 and c_full='0' then
47             z_v := z;
48             p_v := data_from(a);
49             a_rd <= '1';
50             s <= S1;
51             c <= to_data(z_v,10,false);
52             c_wr <= '1';
53             z <= p_v;
54             state <= R2;
55           elsif a_empty='0' and is_eos(a) and s=S1 and c_full='0' then
56             a_rd <= '1';
```

```
57              s <= S0;
58              c <= eos(10);
59              c_wr <= '1';
60              state <= R3;
61            end if;
62          when R1 =>
63              a_rd <= '0';
64              c_wr <= '0';
65              state <= R0;
66          when R2 =>
67              a_rd <= '0';
68              c_wr <= '0';
69              state <= R0;
70          when R3 =>
71              a_rd <= '0';
72              c_wr <= '0';
73              state <= R0;
74        end case;
75      end if;
76    end process;
77 end FSM;
```

Lines 1-6 are for `library` and lines 8-19 for `entity` declarations.

The `architecture` of the actor is defined in lines 21-81. The `state` and `s` variables in lines 22-25 have already been described in section 4.2.2, the variable `z` in line 26 is the VHDL translation of variable `z` in CAPH. In lines 29-30, the variables `v_z` and `p_z` are translation of `v` and `z` variables in CAPH transition rule (line 9 of CAPH code).

Lines 37-76 are the conversion of the CAPH transition rules into a synchronized finite state machine in VDHL.

Lines 40-46 are the translation of the first rule. The `if` condition starts by checking the availability of data on input FIFO. If data is available then it should be a control token indicating a line. Simultaneously, it checks the value of `s`, which must be `S0` here, and whether data can be written on output FIFO i.e. whether the FIFO is not full. If all these conditions are met then the read signal for the input FIFO and the write signal for the output FIFO are asserted, and the start of line control token is written to the output. The variable `z` is also initialized to zero and the variable `s` is changed to `S1` to move to next rule. The `state` variable is assigned `R1`, to change to zero read (resp. write) signals to input (resp. output) FIFOs in the next clock cycle.

Lines 47-55 correspond to the second transition rule. The changes from the previous `if` condition are a check for data at the input instead of "start of line" control token and the fact that variable `s` should now be `S1`. If these conditions are met, the input value is saved in variable `p_v`, the value of the variable `z` is saved in temporary variable `z_v` which is then written at output and `z` is updated with `p_v`, `s` will not change and `state` is assigned `R2` to change FIFO read/write signals.

Lines 56-61 give the generated VHDL code for the third rule. They are similar to the ones given for the first rule, the only difference being that the "end of line" control token is read at input instead of "start of line". The variable `s` is changed to `S0` to move back to the first rule to start processing the next line.

The generated VHDL code when compiled for an FPGA[2], uses 50 Logical Elements (LEs), none of the other resources (memory bit, DSP block etc.) are used. The design operates at the maximum frequency of 250 MHz. The RTL diagram of the generated VHDL code is given in Fig. 5.2. The *state machine* in represented by yellow box in the diagram. The variable `z` is a *register*. The rest of the diagram consists of *equal* operators, *multiplexers* and *registers*. The registers are used to store results before sent at output.

5.3 One-line delay

The *d1l* actor described in this section delays each image of its input stream by one line. For example – taking here 2x2 image for simplicity – if the input stream is

< <1 2> <3 4> > < <5 6> <7 8> >...

then the output stream will be

< <0 0> <1 2> > < <0 0> <5 6> >....

Functionally speaking :

$$
\begin{array}{l}
d1l :< \\
\quad < P_{11}, P_{12}, ..., P_{1n} > \\
\quad < P_{21}, P_{22}, ..., P_{2n} > \\
\qquad \vdots \\
\quad < P_{m1}, P_{m2}, ..., P_{mn} > \\
>
\end{array}
\quad \longrightarrow \quad
\begin{array}{l}
< \\
\quad < 0, 0, ..., 0 > \\
\quad < P_{11}, P_{12}, ..., P_{1n} > \\
\qquad \vdots \\
\quad < P_{(m-1)1}, P_{(m-1)2}, ..., P_{(m-1)n} > \\
>
\end{array}
$$

It works by sending zeros to the output for the first line while saving the input line in an array. For the next lines of the image, the values stored in the array are sent to the output and replaced by those read on input. This continues until the end of the image. The last line stored in the array is discarded. The CAPH code for this example is given in the listing 5.6.

Listing 5.6: CAPH code for one-line delay example

```
actor d1l ()
  in (a:pixel dc)
  out (c:pixel dc)
var s : {S0,S1,S2,S3,S4}=S0
var z : pixel array[256] = [ 0 : 256 ]
var i : unsigned<8>
rules (s, a, z, i) -> (s, c, z, i)
| (S0, '<, _, _) -> (S1, '<, _, _)
| (S1, '<, _, _) -> (S2, '<, _, 0)
| (S2, '>, _, _) -> (S3, '>, _, _)
| (S2, 'p, z, i) -> (S2, '0, z[i-p], i+1)
| (S3, '>, _, _) -> (S0, '>, _, _)
| (S3, '<, _, _) -> (S4, '<, _, 0)
| (S4, 'p, z, i) -> (S4, 'z[i], z[i-p], i+1)
| (S4, '>, _, _) -> (S3, '>, _, _)
;
```

[2]using Quartus toolset for Stratix EP1S60 FPGA

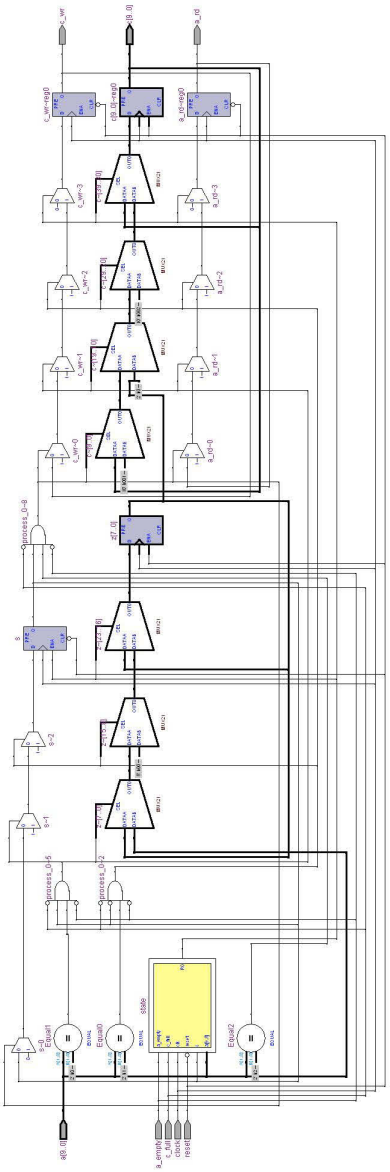


Figure 5.2: RTL view of d1p actor

In the `variable` declaration part, an array is declared. This array will memorize one entire line of the input image. Its length is here fixed to 256 but it can be changed according to the actual width of the input image.

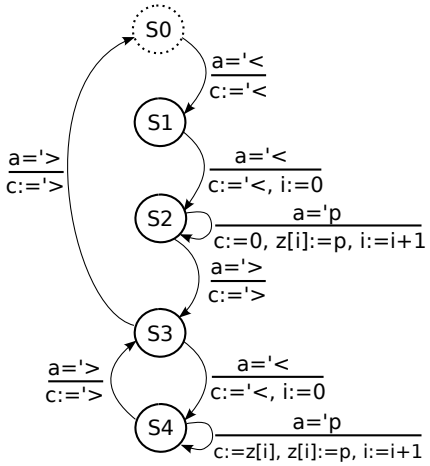


Figure 5.3: One-line delay actor state diagram

The behavior of the actor requires ten transition rules, which can be summarized graphically in Fig. 5.3. The first rule (line 8) reads the "start of image" control token, writes the same token to the output and changes `state` to `S1`. The second rule (line 9) handles the "end of image" control token. The third rule (line 10) handles the "start of line" control token for the first line of the image. The corresponding action writes the "start of line" control token to the output, initializes the `i` variable to zero and moves to state `S2`. In state `S2`, the rule to be fired depends on the input token. If the input token is "`>`" which means "end of line", the same token is written to the output and state is changed to `S3`. If this token is a data token, the fifth rule is fired : we read an input data, write it in the array, increment the array index by one and write zero to the output. The execution of the sixth or seventh rule also depends to the input token. If this token marks the end of image, after writing "`>`" to the output, the state is changed to `S0` to prepare for processing of the next image. If this token marks the start of a new line, the "`<`" token is written to the output, the variable `i` is initialized to zero and `state` is changed to `S4`. The second last rule in line 15 is fired for each data token within a line. It reads the input value, writes the value stored at index `i` of the array `z` to the output, updates the array cell with the read value and increments the index `i`. The last rule in line 16 is fired when an "end of line" control token is read and `state` is moved to `S3` to start waiting for "start of line" control token or "end of image" control token.

The VHDL file generated for this actor is given in listing 5.7. We will focus here on the translation of the code involving the manipulation of array `z`.

Listing 5.7: VHDL code generated for d1l actor

```
1 library ieee;
2 library dc;
3 use ieee.std_logic_1164.all;
4 use dc.dcflow.all;
5 use ieee.std_logic_unsigned.all;
6
7 entity d1l_act is
```

```
8     port (
9      a_empty: in std_logic;
10     a: in std_logic_vector(9 downto 0);
11     a_rd: out std_logic;
12     c_full: in std_logic;
13     c: out std_logic_vector(9 downto 0);
14     c_wr: out std_logic;
15     clock: in std_logic;
16     reset: in std_logic
17     );
18 end d1l_act;
19
20 architecture FSM of d1l_act is
21     type t_state is (R0,R1,R2,R3,R4,R5,R6,R7,R8,R9);
22     type t_enum1 is (S0,S1,S2,S3,S4);
23     type t_z is array(0 to 255) of std_logic_vector(7 downto 0);
24     signal state: t_state;
25     signal i : std_logic_vector(7 downto 0);
26     signal z : t_z := ( others => "00000000");
27     signal s : t_enum1;
28 actorbegin
29   actorprocess(clock, reset)
30     variable p_v : std_logic_vector(7 downto 0);
31     variable i_v : std_logic_vector(7 downto 0);
32   begin
33     if (reset='0') then
34       state <= R0;
35       s <= S0;
36       a_rd <= '0';
37       c_wr <= '0';
38     elsif rising_edge(clock) then
39       case state is
40         when R0 =>
41           if a_empty='0' and is_sos(a) and s=S0 and c_full='0' then
42             a_rd <= '1';
43             s <= S1;
44             c <= sos(10);
45             c_wr <= '1';
46             state <= R1;
47           elsif a_empty='0' and is_eos(a) and s=S1 and c_full='0' then
48             a_rd <= '1';
49             s <= S0;
50             c <= eos(10);
51             c_wr <= '1';
52             state <= R2;
53           elsif a_empty='0' and is_sos(a) and s=S1 and c_full='0' then
54             a_rd <= '1';
55             s <= S2;
56             c <= sos(10);
57             c_wr <= '1';
58             i <= "00000000";
59             state <= R3;
60           elsif a_empty='0' and is_eos(a) and s=S2 and c_full='0' then
61             a_rd <= '1';
62             s <= S3;
63             c <= eos(10);
64             c_wr <= '1';
```

```
65              state <= R4;
66            elsif a_empty='0' and is_data(a) and s=S2 and c_full='0' then
67              i_v := i;
68              p_v := data_from(a);
69              a_rd <= '1';
70              s <= S2;
71              c <= to_data("00000000",10,true);
72              c_wr <= '1';
73              z(conv_integer(i_v)) <= p_v;
74              i <= i_v + "00000001";
75              state <= R5;
76            elsif a_empty='0' and is_eos(a) and s=S3 and c_full='0' then
77              a_rd <= '1';
78              s <= S0;
79              c <= eos(10);
80              c_wr <= '1';
81              state <= R6;
82            elsif a_empty='0' and is_sos(a) and s=S3 and c_full='0' then
83              a_rd <= '1';
84              s <= S4;
85              c <= sos(10);
86              c_wr <= '1';
87              i <= "00000000";
88              state <= R7;
89            elsif a_empty='0' and is_data(a) and s=S4 and c_full='0' then
90              i_v := i;
91              p_v := data_from(a);
92              a_rd <= '1';
93              s <= S4;
94              c <= to_data(z(conv_integer(i_v)),10,true);
95              c_wr <= '1';
96              z(conv_integer(i_v)) <= p_v;
97              i <= i_v + "00000001";
98              state <= R8;
99            elsif a_empty='0' and is_eos(a) and s=S4 and c_full='0' then
100             a_rd <= '1';
101             s <= S3;
102             c <= eos(10);
103             c_wr <= '1';
104             state <= R9;
105           end if;
106         when R1 =>
107             a_rd <= '0';
108             c_wr <= '0';
109             state <= R0;
110         when R2 =>
111             a_rd <= '0';
112             c_wr <= '0';
113             state <= R0;
114         when R3 =>
115             a_rd <= '0';
116             c_wr <= '0';
117             state <= R0;
118         when R4 =>
119             a_rd <= '0';
120             c_wr <= '0';
121             state <= R0;
```

```
122          when R5 =>
123               a_rd <= '0';
124               c_wr <= '0';
125               state <= R0;
126          when R6 =>
127               a_rd <= '0';
128               c_wr <= '0';
129               state <= R0;
130          when R7 =>
131               a_rd <= '0';
132               c_wr <= '0';
133               state <= R0;
134          when R8 =>
135               a_rd <= '0';
136               c_wr <= '0';
137               state <= R0;
138          when R9 =>
139               a_rd <= '0';
140               c_wr <= '0';
141               state <= R0;
142        end case;
143      end if;
144    end process;
145 end FSM;
```

A variable `z` of type array is declared in line 27, where `t_z` corresponds to the earlier declared array type in line 24.

The translation of the CAPH fifth rule is in lines 67-76. The `if` condition checks the availability of input (resp. output) FIFO for reading (resp. writing), the type of the corresponding token and the value of `s`. When the condition is true, the action is executed. It first stores the value of index variable `i` in `i_v`, asserts the read signal for the input FIFO and stores the input data in variable `p_v`. In the same way, the write signal for the output FIFO is asserted and zero is written to the output. The value stored in `p_v` is saved at index `i_v` of array `z`. The array index `i` is incremented by one. The `s` variable is changed to `S2`. Finally, `state` is moved to `R5` to change the read/write signals of the FIFOs.

The ninth rule of CAPH is translated in lines 90-99. The `if` condition is similar to the one described above. The array index `i` is stored in `i_v`. The read signal to the input FIFO is asserted and the read value is stored in `p_v`. The write signal of the output FIFO is asserted and the value at index `i_v` of the array `z` is written to the output. Simultaneously, the value at the same index `i_v` of array is updated with `p_v`.

The generated VHDL code when compiled for an FPGA[3], uses 140 LEs and 2048 bits of memory. The design operates at the maximum frequency of 140 MHz. The RTL diagram of the generated VHDL code is given in Fig. 5.4. It is important to note that this description uses memory for implementing the array `z` and not logic elements (in Fig. 5.4 `z` is a synchronous block RAM). This is achieved by accessing the array elements in the finite state machine in a way[4] which will infer a block RAM[5] for the array `z` on the FPGA. The resulting RAM block will consist of a read_address, write_address, input, output, clock and write_enable. Note also

[3] using Quartus toolset for Stratix EP1S60 FPGA
[4] By following the coding styles provided by tool vendors
[5] with a depth of 256 and 8 bits width

that the array `z` is both read and written in the same clock cycle, this is achieved by sending read and write addresses to the RAM on each clock cycle.

5.4 1x3 Convolution

Convolution is a mathematical operation which is fundamental for many image processing operations. It generally provides a way of multiplying an image with an array. This array is also known as a kernel. This example will demonstrate the implementation of a 1x3 convolution operation in CAPH. Here 1x3 refers to the dimensions of the kernel. The mathematical formulation of a 1x3 convolution operation is :

$$
\begin{array}{lll}
conv1x3 :< & & < \\
\quad < P_{11}, P_{12}, ..., P_{1n} > & & \quad < f(P_{11}), f(P_{12}), ..., f(P_{1n}) > \\
\quad < P_{21}, P_{22}, ..., P_{2n} > & \longrightarrow & \quad < f(P_{21}), f(P_{22}), ..., f(P_{2n}) > \\
\quad \vdots & & \quad \vdots \\
\quad < P_{m1}, P_{m2}, ..., P_{mn} > & & \quad < f(P_{m1}), f(P_{m2}), ..., f(P_{mn}) > \\
> & & >
\end{array}
$$

$$
\begin{aligned}
&Where\ f(P_{i,j}) = (k0 * P_{i-2,j} + k1 * P_{i-1,j} + k2 * P_{i,j}) >> n \\
&and\ P_{-1,j} = P_{-2,j} = 0
\end{aligned}
$$

The implementation given here uses the `d1p` operator introduced in section 5.2[6]. The CAPH code for this example is given in listing 5.8.

Listing 5.8: CAPH code for 1x3 convolution example

```
1  const k=[1,2,1];
2  const norm=2;
3
4  type uint = unsigned<8>;
5
6  actor d1p ()
7    in (a:uint dc)
8    out (c:uint dc)
9   ...
10
11 actor maddn (k:uint array[3], n:uint)
12   in (a:uint dc, b:uint dc, c:uint dc)
13   out (s:uint dc)
14 rules (a,b,c) -> s
15 | ( '<,'<,'<) -> '<
16 | ( 'zzp, 'zp, 'p) -> '(k[0]*zzp+k[1]*zp+k[2]*p)>>n
17 | ( '>,'>,'>) -> '>
18 ;
19
20 stream x:uint dc from "sample.txt";
21 stream r:uint dc to "result.txt";
```

[6]It is also possible to give a formulation of convolution actor using a single actor

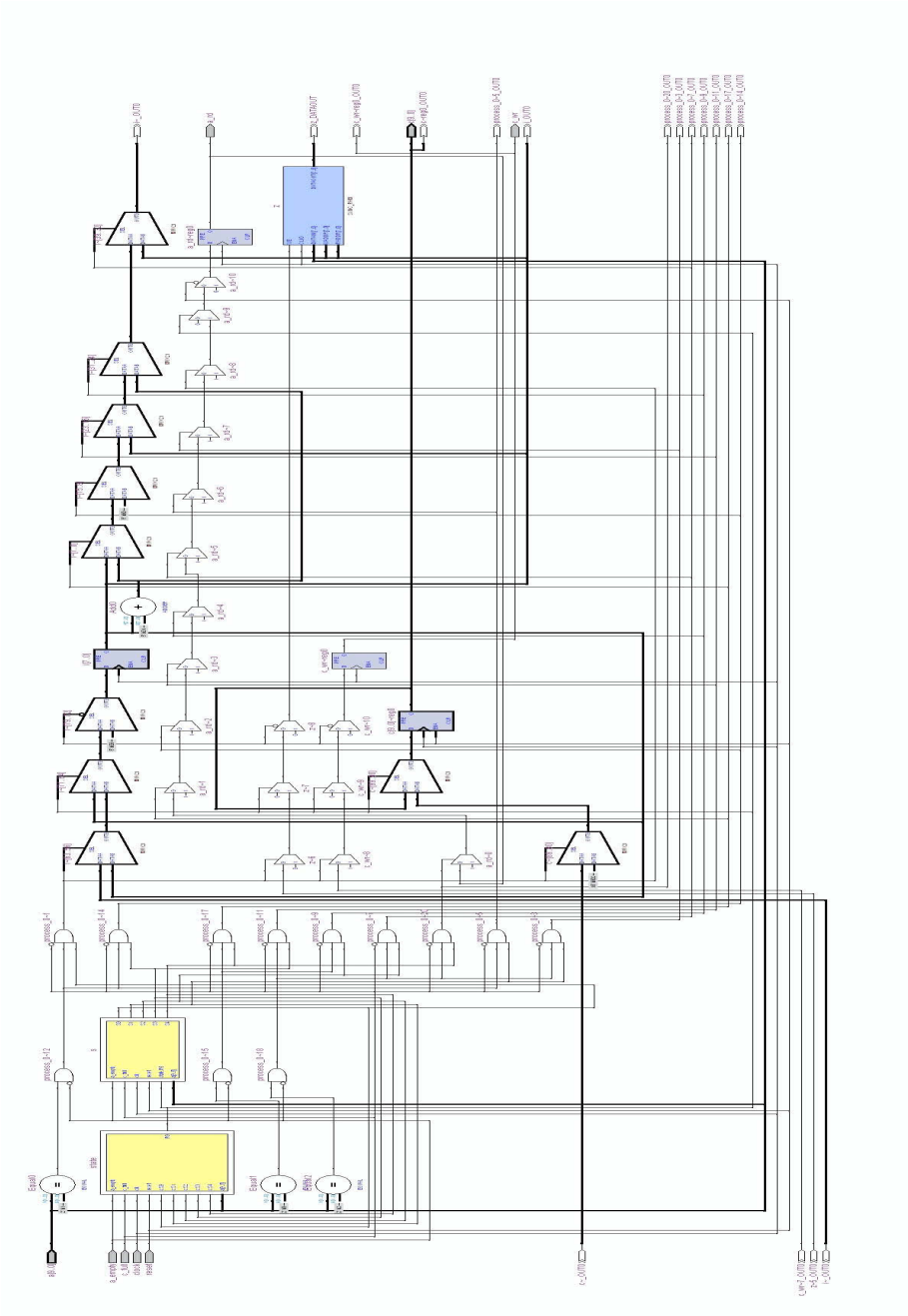


Figure 5.4: RTL view of d1l actor

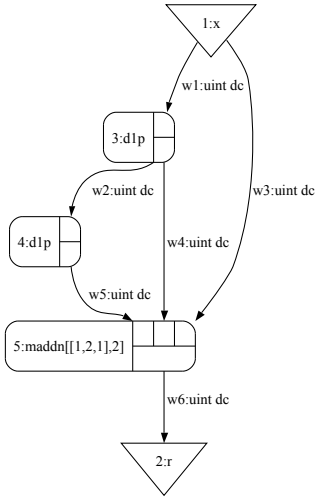


Figure 5.5: Dataflow graph of 1x3 Convolution example

```
22
23 net xz = d1p x;
24 net xzz = d1p xz;
25 net r = maddn [k,norm] (xzz, xz, x);
```

Lines 1-2 define two constants : `k` and `norm`. Here `k` is the kernel array and `norm` is the normalization factor, given here as a right shifting factor.

Line 4 introduces a type abbreviation.

Lines 6-9 give the excerpt of `d1p` actor, it has been already described and commented in section 5.2.

The `maddn` actor defined in line 11-18 performs the actual convolution operation. It calculates the value of a pixel by multiplying a 1x3 neighborhood of the current pixel by the kernel and divide the resulting value by the normalize factor (here right shift operator is performed instead of division).

Lines 23-25 define the dataflow network describing how the convolution operation is performed on each line of the input stream. For this, two data streams, `xz` and `xzz` are constructed using the `d1p` actor. Finally, the resulting three streams (`x`, `xz`, `xzz`), constituting the 1x3 neighborhood of each input pixel, are combined by the `maddn` actor as shown in dataflow graph in Fig. 5.5.

The generated VHDL design when compiled for an FPGA[7], uses 418 LEs, none of the other resources (memory bit, DSP block etc.) are used. The design operates at the maximum frequency of 143 MHz. The RTL view of `maddn` actor is shown in Fig. 5.6. It consists of *state machine*, *multiplexers*, *operators* and *registers* to perform the convolution operation. The RTL view of the 1x3 convolution network file is shown in Fig. 5.7.

5.5 3x3 Convolution

This example will demonstrate the implementation of a 3x3 convolution operation operating on images in CAPH. Here the size of the kernel array is 3x3. The mathematical formulation of

[7]using Quartus toolset for Stratix EP1S60 FPGA

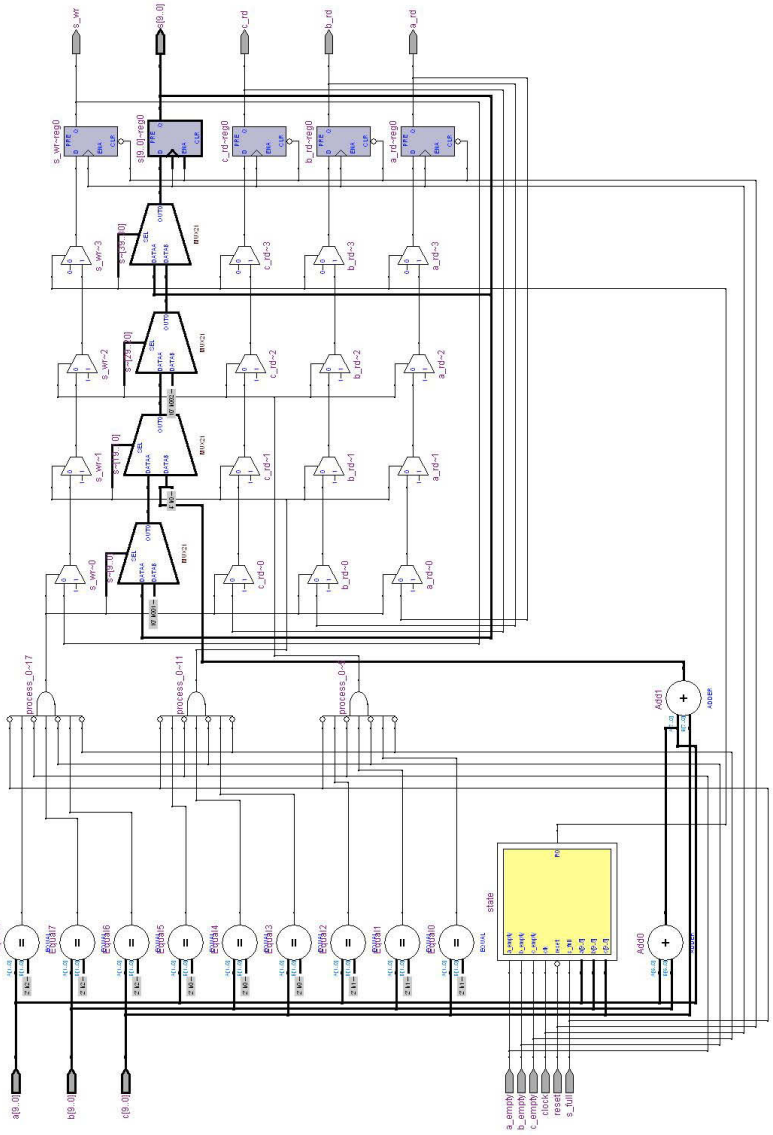


Figure 5.6: RTL view of maddn actor

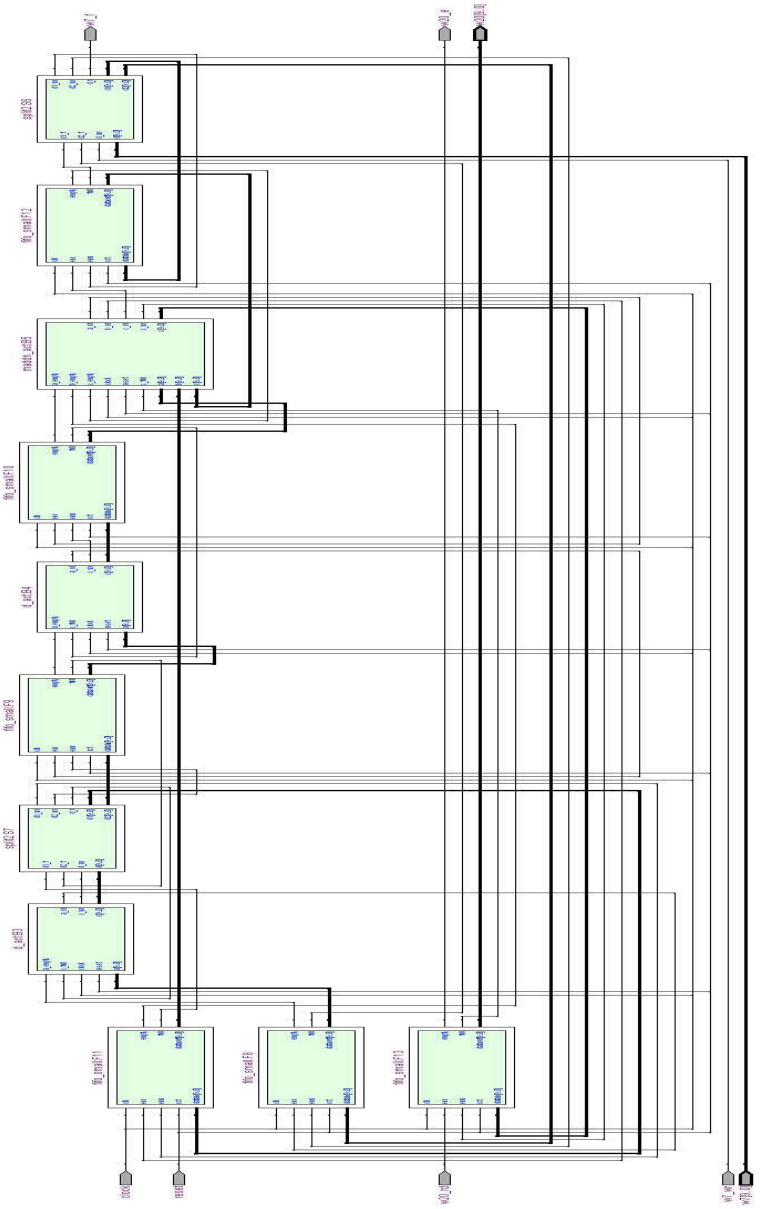


Figure 5.7: RTL view of network file for 1x3 Convolution

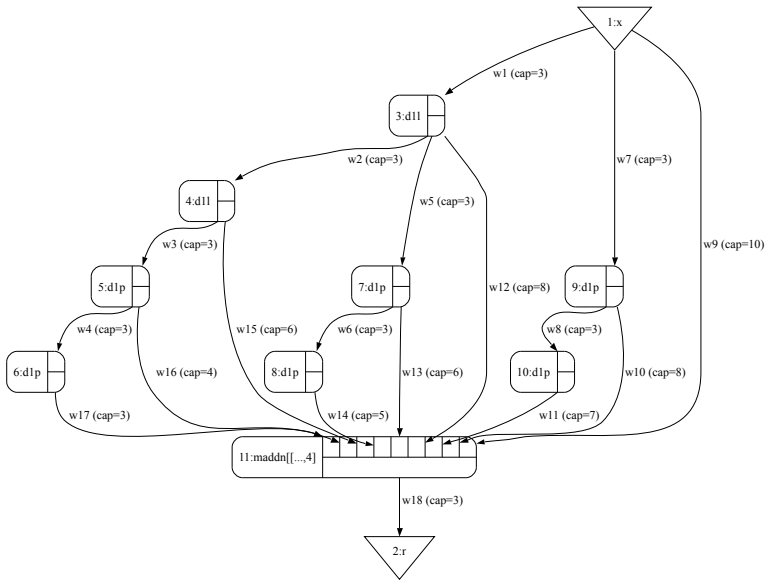


Figure 5.8: Dataflow graph of 3x3 Convolution application

this operation is:

$$
\begin{array}{lll}
conv3\texttt{x}3 :< & & < \\
\quad < P_{11}, P_{12}, ..., P_{1n} > & & \quad < f(P_{11}), f(P_{12}), ..., f(P_{1n}) > \\
\quad < P_{21}, P_{22}, ..., P_{2n} > & \longrightarrow & \quad < f(P_{21}), f(P_{22}), ..., f(P_{2n}) > \\
\quad \vdots & & \quad \vdots \\
\quad < P_{m1}, P_{m2}, ..., P_{mn} > & & \quad < f(P_{m1}), f(P_{m2}), ..., f(P_{mn}) > \\
> & & >
\end{array}
$$

$$
\begin{aligned}
Where\ f(P_{i,j}) = (&k0 * P_{i-2,j-2} + k1 * P_{i-2,j-1} + k2 * P_{i-2,j} \\
&+ k3 * P_{i-1,j-2} + k4 * P_{i-1,j-1} + k5 * P_{i-1,j} \\
&+ k6 * P_{i,j-2} \quad + k7 * P_{i,j-1} \quad + k8 * P_{i,j} \quad) >> n \\
and\ P_{-2,j} = P_{-1,j} &= P_{i,-2} = P_{i,-1} = 0
\end{aligned}
$$

We take the same approach to that in section 5.4 for the 1x3 convolution : a set of delayed streams is first constructed using one-pixel and one-line delay actors and these streams are combined using a multiply, add and normalize (`maddn`) actor. The dataflow graph of the application is shown in Fig. 5.8. The CAPH code for this example is given in listing 5.9.

Listing 5.9: CAPH code for 3x3 convolution example

```
1 const k = [1,2,1, 2,4,2, 1,2,1];
2 const norm=4;
3
4 type uint = unsigned<16>;
5
6 actor d1p ()
```

```
7    in (a:uint dc)
8    out (c:uint dc)
9   ...
10
11 actor d1l ()
12   in (a:uint dc)
13   out (c:uint dc)
14  ...
15
16 actor maddn (k:uint array[9], n:uint)
17   in  (x0:uint dc, x1:uint dc, x2:uint dc, x3:uint dc,
18         x4:uint dc, x5:uint dc, x6:uint dc, x7:uint dc, x8:uint dc)
19   out (s:uint dc)
20 rules ( x0, x1, x2, x3, x4, x5, x6, x7, x8) -> s
21 | ('<, '<, '<, '<, '<, '<, '<, '<, '<) -> '<
22 | ( 'x0, 'x1, 'x2, 'x3, 'x4, 'x5, 'x6, 'x7, 'x8) ->
23     '(k[0]*x0+k[1]*x1+k[2]*x2+k[3]*x3+k[4]*x4+k[5]*x5+k[6]*x6+k[7]*x7+k[8]*x8)>>n
24 | ('>, '>, '>, '>, '>, '>, '>, '>, '>) -> '>
25 ;
26
27 stream x:uint dc from "sample.txt";
28 stream r:uint dc to "result.txt";
29
30 net neigh13 x =
31  let z = d1p x in
32  x, z, d1p z
33 ;
34
35 net neigh33 x =
36   let xz = d1l x in
37   let xzz = d1l xz in
38   neigh13 x, neigh13 xz, neigh13 xzz
39 ;
40
41 net ((y11,y12,y13),(y21,y22,y23),(y31,y32,y33))=neigh33 x;
42
43 net r=maddn [k,norm](y33,y32,y31,y23,y22,y21,y13,y12,y11);
```

The first part of the code defines `constants` and `types` in lines 1-4. As described earlier, here the kernel array `k` consist of nine elements.

The `d1p` and `d1l` actors in lines 6-14 have been described in section 5.2 and 5.3 respectively.

Here the `maddn` actor in lines 16-25 has nine input values instead of three as in 1x3 convolution example.

The main difference is in the network declaration part (lines 30-43), where the nine pixels belonging to the neighborhood of the current pixel are obtained from the input data stream and given as input to the `maddn` actor. This is achieved by declaring two *higher-order wiring functions*, `neigh13` and `neigh33`. The `neigh13` function encapsulates the stream formulation process already seen in the 1x3 convolution example : it takes an input stream and generates 3 output streams corresponding to the input stream : the first is same input stream, the second is this input stream delayed by one pixel and the third is the same input stream delayed by 2 pixels. The `neigh33` function does the same thing but delays streams by zero, one and two lines respectively and apply the `neigh13` function to each of the resulting streams. It

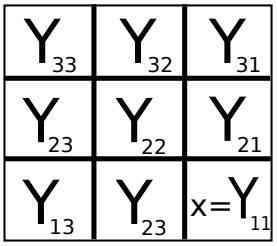


Figure 5.9: Neighborhood of current pixel x

eventually produces nine output streams, Y_{11},Y_{12},Y_{13},Y_{21},Y_{22},Y_{23},Y_{31},Y_{32} and Y_{33} representing the neighborhood of the input stream x as shown in Fig. 5.9. These nine values are given as input to `maddn` actor to calculate the final value.

All the actors are connected to each other through FIFOs to form the dataflow network. The important thing to note here is the different size of each FIFO. Here the size of each FIFO is calculated by SystemC profiling. The code generated by the SystemC back end has been run to calculate the run time occupancy of each FIFO. The resulting values are then used by the VHDL back end to assign a size to each FIFO. This size assigned is shown in dataflow graph of application in Fig. 5.8.

The generated VHDL code when compiled for an FPGA[8], uses 2464 LEs (4% of total), 8192 bits of memory (<1% of total) and no DSP blocks. The memory bits are used by two `d1l` actors. These actors use memory to store one line in array `z` as described in section 5.3. In this example, the size of each pixel is 16 bits and each line consists of 256 pixels, so memory used by one `d1l` actor is 4096 bits (256×16). On the other hand, all the FIFOs are implemented using LEs. The reason is, size of all the FIFOs is small (the biggest is with a capacity of 10, wire **w9** in the Fig. 5.8). The design operates at the maximum frequency of 60 MHz. These results are for 256x256 image and 3x3 kernel array. The RTL diagram of the network file is shown in Fig. 5.10.

Table 5.1 summarizes all the four examples described in sections 5.2 to 5.5. It gives the CAPH lines of code (LOC) to write an example, the VHDL LOC generated by the CAPH compiler and FPGA resources used to execute the generated VHDL code on an FPGA.

Table 5.1: Examples summary

	d1p	**d1l**	**1x3 convolution**	**3x3 convolution**
CAPH LOC	10	17	27	58
VHDL LOC	76	144	285	660
Max Frequency	250 MHz	140 MHz	143 MHz	60 MHz
Logic Elements	50	140 (<1%)	418 (<1%)	2,464 (4%)
Memory Bits	0	2,048 (<1%)	0	8,192 (<1%)

[8]using Quartus toolset for Stratix EP1S60 FPGA

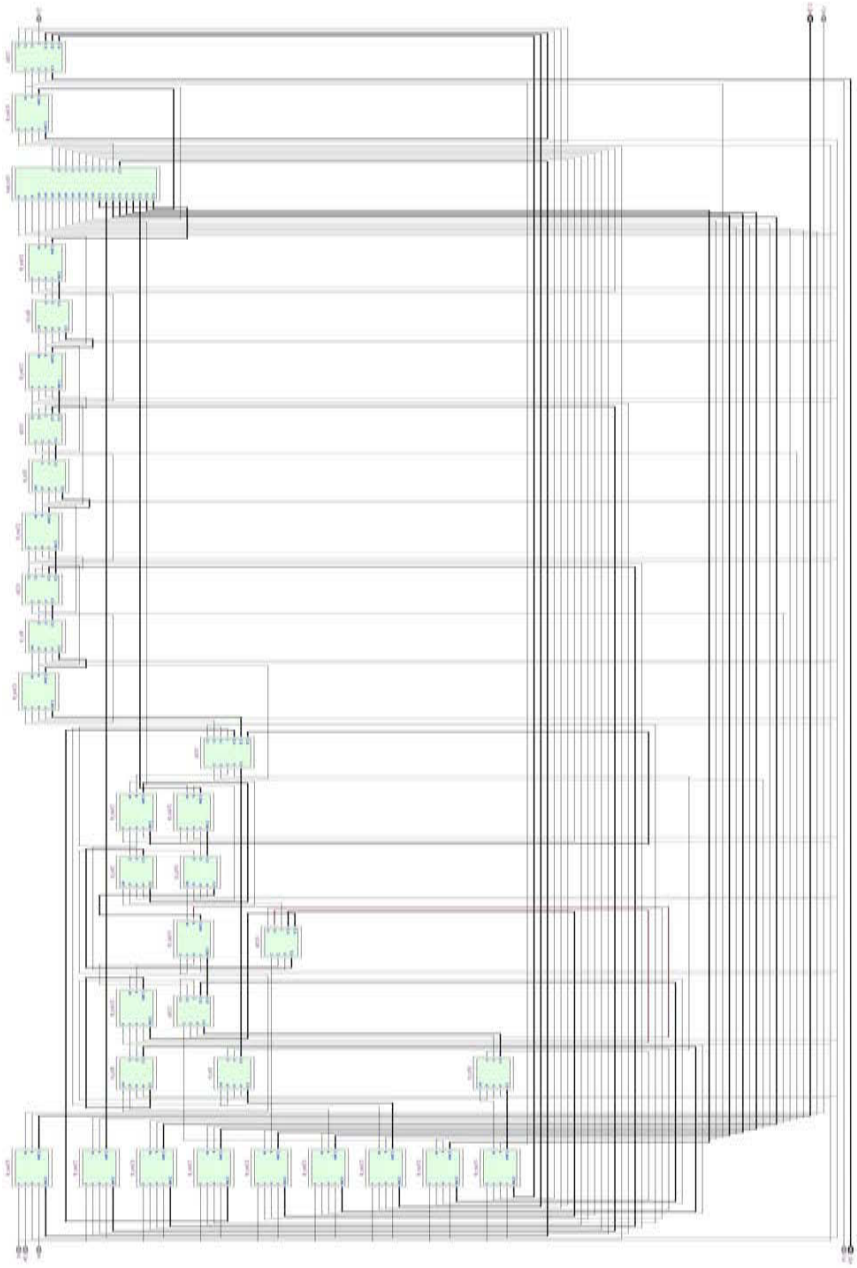


Figure 5.10: RTL view of 3x3 Convolution application

5.6 Functions

There are two types of functions in CAPH : global functions and external functions.

5.6.1 Global Functions

Global functions are typically used to simplify the expression of a program. They will be directly translated to VHDL or SystemC functions by the corresponding back ends.

Example 1

In this example, the input value is scaled by a factor `k` and the result is written to the output. The CAPH implementation for this example using global function is given in listing 5.10.

Listing 5.10: Scale example using function

```
1 function mult (x,y)=  x*y : signed<8> -> signed<8>;
2
3 actor scale (k:signed<8>)
4   in (a:signed<8>)
5   out (c:signed<8>)
6 rules a -> c
7 | p -> (mult(p,k))
8 ;
```

Here instead of directly multiplying input `p` with parameter `k`, the multiplication is performed by calling the function `mult`. This function is declared in the first line of code.

The code generated by the SystemC or VHDL back end for global functions is generated in a separate file. For this example, the VHDL back end generates a file `main_globals.vhd` which contains the code given in listing 5.11.

Listing 5.11: VHDL code generated for mult function

```
1 package main_globals is
2   function mult(x:std_logic_vector; y:std_logic_vector)
3             return std_logic_vector;
4 end main_globals;
5
6 package body main_globals is
7   function mult(x:std_logic_vector; y:std_logic_vector)
8             return std_logic_vector is
9   begin
10    return mul(x,y);
11  end mult;
12 end main_globals;
```

Example 2

This example implements a median filter in CAPH. The code for this example in CAPH is given in listing 5.12.

Listing 5.12: Median filter using functions

```
type uint = unsigned<8>;

function min_fn(x,y) = if x < y then x else y;
function max_fn(x,y) = if x > y then x else y;
function med_fn(x,y,z) = min_fn (max_fn (x,y), min_fn (max_fn (y,z), max_fn(x,z)));

actor median ()
    in (a:uint dc)
  out (c:uint dc)
var s : {S0,S1,S2,S3} = S0
var z : uint
var zz : uint
rules (s, a, z, zz) -> (s, c, z, zz)
| (S0, '<, _, _) -> (S1, '<, 0, 0)
| (S1, '>, _, _) -> (S0, '>, _, _)
| (S1, 'p, z, _) -> (S2, 'p, p, z)
| (S2, 'p, z, _) -> (S3, 'min_fn(z,p), p, z)
| (S3, 'p, z, zz) -> (S3, 'med_fn(zz,z,p), p, z)
| (S3, '>, _, _) -> (S1, '>, _, _)
;

stream x:uint dc from "sample.txt";
stream r:uint dc to "result.txt";

net r = median x;
```

This example uses three functions `min_fn`,`max_fn` and `med_fn`. In fact, the `med_fn` in turn uses the other two functions to calculate the final result. The VHDL backend generates the code for these functions in `median_globals.vhd` which contains the code given in listing 5.13.

Listing 5.13: VHDL code generated for functions

```
package median_globals is
  function med_fn(x:std_logic_vector; y:std_logic_vector;
                  z:std_logic_vector)
             return std_logic_vector;
  function max_fn(x:std_logic_vector; y:std_logic_vector)
             return std_logic_vector;
  function min_fn(x:std_logic_vector; y:std_logic_vector)
             return std_logic_vector;
end median_globals;

package body median_globals is
  function med_fn(x:std_logic_vector; y:std_logic_vector;
                  z:std_logic_vector)
           return std_logic_vector is
  begin
    return min_fn(max_fn(x,y), min_fn(max_fn(y,z), max_fn(x,z)));
  end med_fn;
  function max_fn(x:std_logic_vector; y:std_logic_vector)
           return std_logic_vector is
```

```
20   begin
21     return cond(x > y,x,y);
22   end max_fn;
23   function min_fn(x:std_logic_vector; y:std_logic_vector)
24             return std_logic_vector is
25   begin
26     return cond(x < y,x,y);
27   end min_fn;
28 end median_globals;
```

5.6.2 External Functions

CAPH also provides a way to call functions written in VHDL or SystemC. These functions must be provided in a file `extfns.vhdl` (resp. `extfns.cpp`). The example will describe it in CAPH.**Example**

Here the same scale example described in section 5.6.1, will be implemented through external function. The CAPH code for this implementation is given in listing 5.14.

Listing 5.14: Scale example using external function

```
1 function mult = extern "mult_c", "mult_vhdl", "mult_ml" :
2                  signed<8> * signed<8> -> signed<8>;
3
4 actor scale (k:signed<8>)
5   in  (a:signed<8> )
6   out (c:signed<8> )
7 rules a -> c
8 | p -> (mult(p,k))
9 ;
```

The `mult_c`, `mult_vhdl` and `mult_ml` in the above code mean the implementation of `mult` function in SystemC, VHDL and Ocaml respectively. The programmer can also use the same name for all three implementations. It is also programmer's responsibility to ensure the type compatibility of the implemented function with the type provided in function declaration. The requirement for the external function to inter-operate with the code generated by the backend or the simulator are detailed in the CAPH language reference manual [3].

Chapter 6

Applications

This chapter aims at demonstrating the effectiveness of CAPH in a realistic context. For this, we describe the implementation of three applications : *motion detection*, *connected component labeling* and *JPEG encoding*. For each application we first describe the objectives and principles of the algorithms used, then how they are formalized in CAPH and finally the results obtained by the derived FPGA implementation. For the last application we also include a comparison of the obtained results with a handcrafted VHDL solution and another dataflow language (CAL). The execution of code on a FPGA platform is not as simple as just compile and execute. An important issue is the interaction with I/O devices. The first section of this chapter there gives information about how the VHDL code generated by CAPH is interfaced to physical I/O devices on a typical FPGA platform.

6.1 Compiling CAPH Programs on FPGA

The platform used in this thesis to validate experimental results, is a smart camera SeeMOS, developed at LASMEA since 2004[84]. The primary objective of the SeeMos Project is to develop a research platform dedicated to active vision and in particular to the early vision process. The SeeMOS camera is an embedded system composed of a modular hardware architecture. It

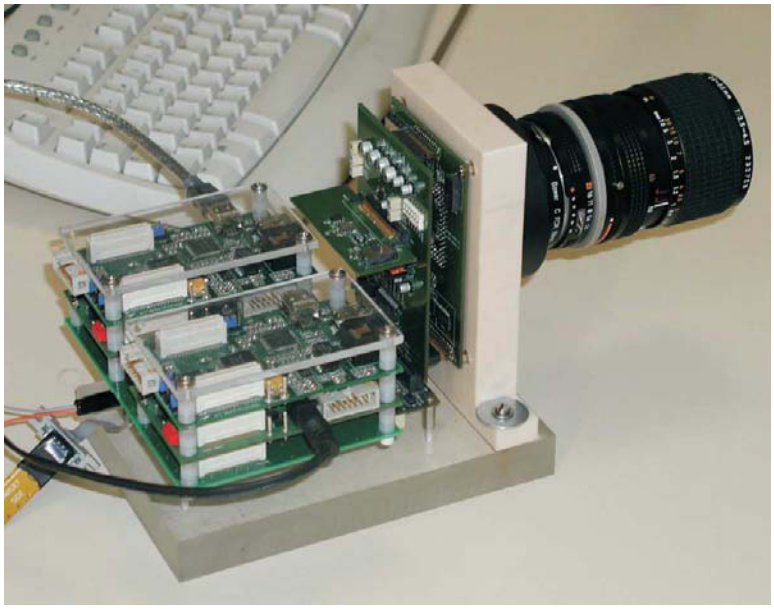


Figure 6.1: SeeMOS smart camera

(figs. 6.1 and 6.2) consists of :

- CMOS imager ;
- Inertial device ;
- FPGA ;
- 5 external SRAM memory blocks ;
- DSP co-processing device ;
- Interface for firewire communication.

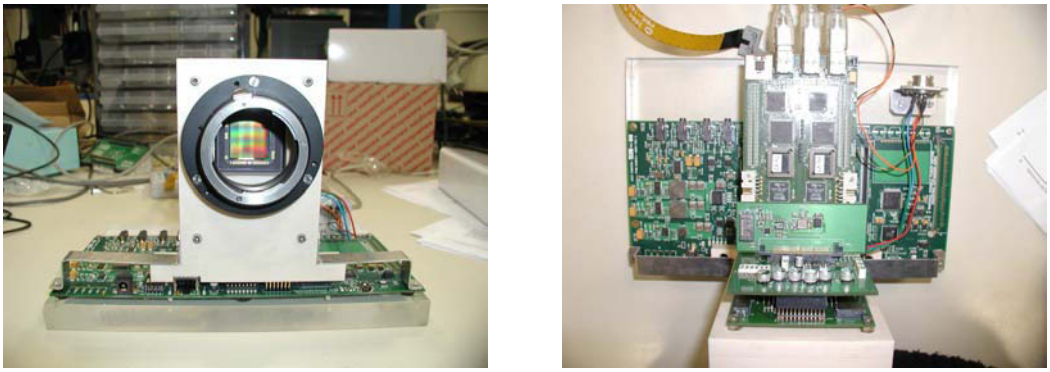


Figure 6.2: SeeMOS camera, developed at LASMEA

The CMOS imager is a LUPA-4000 model by Cypress Semiconductor [85]. It is a monochrome image sensor with 4 Mpixels resolution (2048 x 2048). It can acquire up to 66 Mpixels per second, each pixel coded with 10 bits. The acquisition is done in the "global shutter" mode, to avoid the problems of "rolling shutter" mode in certain CMOS imagers [86]. The acquisition frequency allows a frame rate of more than 200 frames per second in VGA mode (640 x 480), in good lighting conditions.

The CMOS imager is selected due to its ability to randomly address pixels within an image. This feature is particularly useful in motion detection applications where only the part of image containing object is required. The random addressing makes it possible to combine different parts of an image by addressing only a small part of a photosensitive sensor, even for high resolution at high acquisition speed. The design of the SeeMOS camera fully exploits random addressing, for example a speed of 1000 fps can be obtained for 140 x 140 resolution images.

The inertial device consists of three accelerometers aligned at the X, Y and Z axes of a sensor and three gyroscopes. The inertial data estimates the 3D movement of the camera (ego-motion) and also its orientation and position.

All parts of the platform are connected through the FPGA (fig. 6.3) which is an Altera Stratix model EP1S60F1020C7 and acts as the central part of the system. It is responsible for connection, control and synchronization of sensor modules (imager + interial device), external RAMs, the communication card (firewire interface) and the co-processor (DSP card). The characteristics of the FPGA used are detailed in table 6.1.

The FPGA card (fig. 6.3) is also connected to 5 blocks of static RAM. Each block has a capacity of 2 MB and has private data and address buses. So, different blocks can be accessed concurrently. This is useful for exploiting parallelism offered by the FPGA device.

The interface between the camera and host system is accomplished by firewire module (IEEE 1394). It offers a bandwidth of 20 Mbytes per second from the camera to the external environment (host system), and 10 Mbytes in the other direction. These rates are achievable i.e. these rates can be reached when transferring data between camera and host system.

One important feature of the SeeMOS hardware platform is its modularity as the different hardware parts are on different card (fig. 6.4). This facilitates upgrading the platform, as cards can be easily replaced. One part of the platform can be changed or updated without affecting the other parts and the overall connectivity. Thanks to this property, the platform has evolved

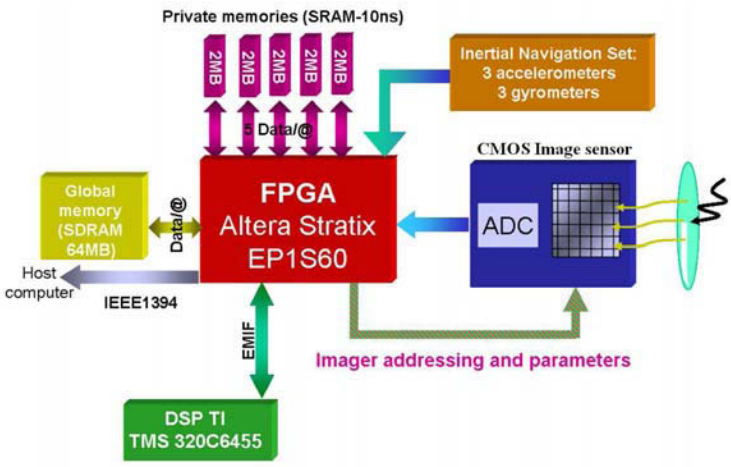


Figure 6.3: Hardware architecture of the SeeMOS platform

Model	**Altera Stratix EP1S60F1020C7**
LEs (Logic Elements)	57.120
M512 RAM blocks (32 x 18 bits)	574
M4K RAM blocks (128 x 36 bits)	292
M-RAM blocks (4K x 144 bits)	6
Total RAM bits	5.215.104
DSP blocks	18
Embedded multipliers (9 x 9-bit)	144
PLLs (Phase-Locked Loops)	12
Package	1.020-Pin FineLine BGA
User I/O pins	773
Pitch (mm)	1,00
Area (mm^2)	1.089
Length x width (mm x mm)	33 x 33
Speed grade	-7

Table 6.1: Characteristics of the FPGA device used in SeeMOS smart camera

constantly, as shown in 6.1 and 6.2. Future proposed changes on the platform are, integration of a dedicated DSP co-processor (instead of DSK Texas) and replacement of the FPGA with a new generation device (Stratix III or Stratix IV).

The above characteristics make the SeeMOS smart camera a good research platform to capture and process images. On the other hand, the number of different hardware elements and their heterogeneity raises problems for the utilization of these resources. The programming of such a platform requires expertise at the hardware level and also knowledge of different programming language and development environments. As a result, the implementation of an application on the SeeMOS platfrom can be a challenging task. One of the motivation of this thesis is to make this easier by the use of the CAPH language.

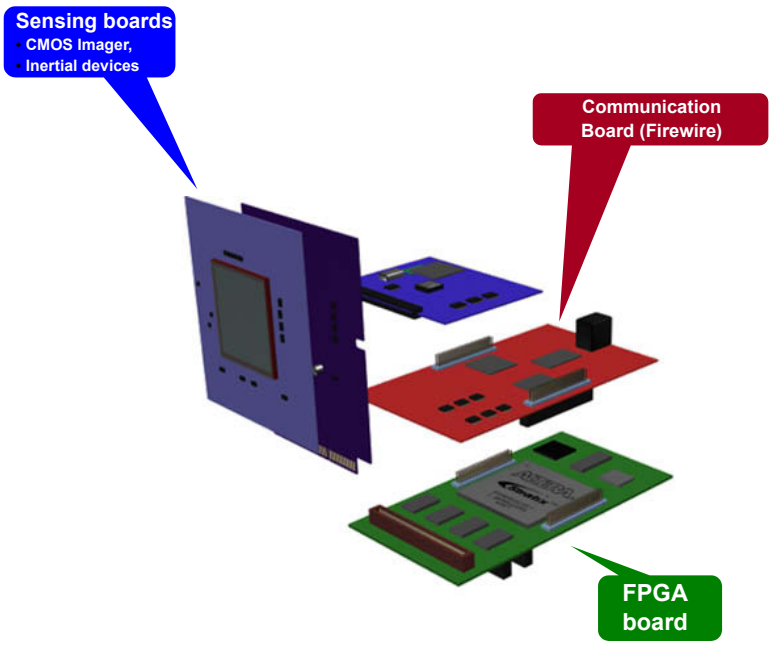


Figure 6.4: Different cards forming the heterogeneous SeeMOS platform

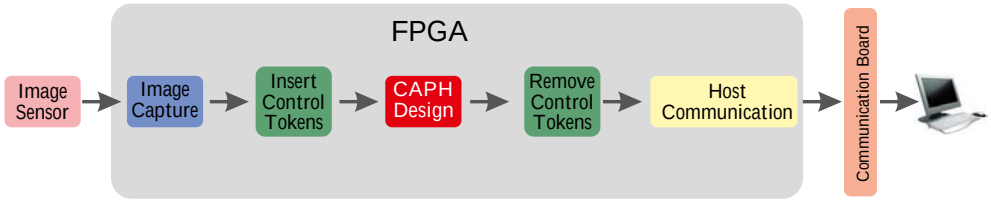


Figure 6.5: FPGA I/O

To test our applications, the design generated by CAPH is inserted into a "template architecture" developed in VHDL. This template takes care of the drivers to control the imager and other sensors, and also the communication with host computer. In addition to these two IPs, two other IPs are responsbile for insertion (resp. removal) of control tokens before (resp. after) the CAPH design as shown in fig. 6.5. The detail of the token insertion process, for an input stream of 8×8 image is shown in fig. 6.6.

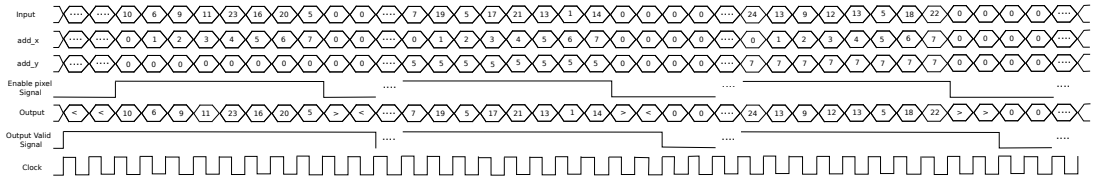


Figure 6.6: Structured stream generation for 8x8 image

The token removal process is the same as described in section 4.1.3. It sends pixels to the host computer for display, the user being responsible for adjusting the `width` and `height` parameters in the C++ application to display the corresponding image on the screen.

The complete design comprising of the VHDL code produced by the CAPH program and supporing IPs is compiled and downloaded to the FPGA using the Altera Quartus toolset[1].

[1]The code generated by CAPH is generic and does not include any specific parameters for any particular type of FPGA. It can be compiled by any tool. In fact, the last experiment in this chapter has also been compiled by the Xilinx ISE design tool.

6.2 Motion Detection Application

6.2.1 Objective

The objective of this application is to detect the presence of a moving object in a scene. A moving object is detected by spatio-temporal changes in the grey-level representation of successive frames. The algorithm estimates temporal variations by calculating the grey-level difference between two consecutive frames. As a result, a rectangular window is drawn surrounding the detected object in motion in the scene.

6.2.2 Principle

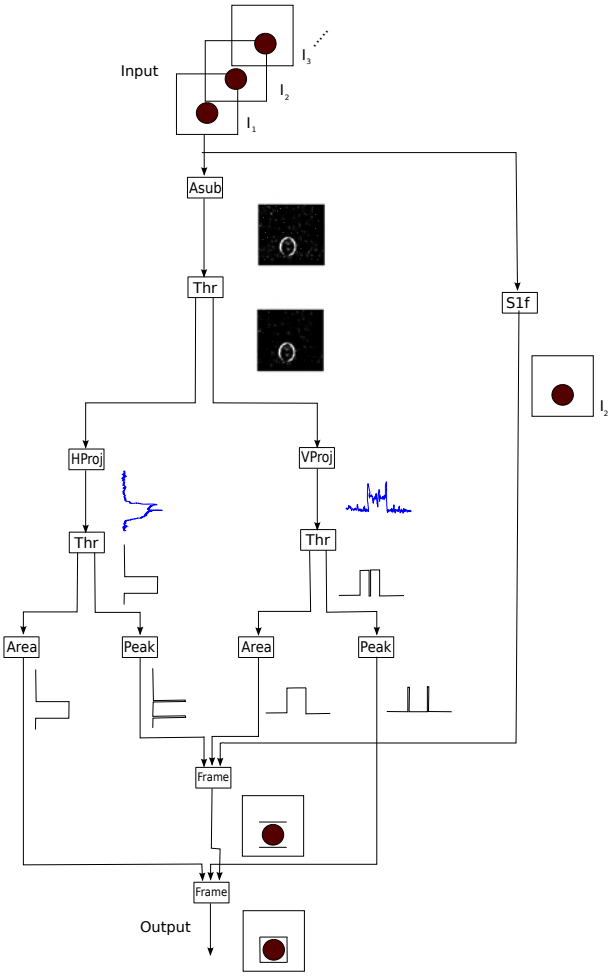


Figure 6.7: Different steps of motion detection application

The different steps of the algorithm, as illustrated in figure 6.7, are

– Computing the image difference between two consecutive frames by calculating the pixel by pixel difference. This is implemented in CAPH with the `asub` actor.

– Thresholding this difference to remove the noise and to obtain a binary image. This is implemented in CAPH with the `thr` actor.

- Computing the horizontal projection on this binary image. This projection is the sum of values of all columns, calculated independently for each line. This is implemented in CAPH with the `hproj` actor.
- Thresholding this projection to extract horizontal band(s) where moving object is likely to be found. This is implemented in CAPH with the `thr` actor.
- Computing the vertical projection (column-wise sum) on the binary image. This projection is the sum of values of all lines, calculated independently for each column. This is implemented in CAPH with the `vproj` actor.
- Thresholding this projection to extract vertical band(s) where a moving object is likely to be found. This is implemented in CAPH with the `thr` actor.
- Computing the peaks of the thresholded horizontal band(s). This is implemented in CAPH with the `peak` actor.
- Computing the whole area containing the thresholded horizontal band(s). This is implemented in CAPH with the `area` actor.
- Computing the peaks of the thresholded vertical band(s). This is implemented in CAPH with the `peak` actor.
- Computing the whole area containing the thresholded vertical band(s). This is implemented in CAPH with the `area` actor.
- Drawing the horizontal lines of a rectangular window on the next frame of the input image. This is implemented in CAPH with the `frame` actor. The next frame is obtained by the `s1f` actor.
- Finally, drawing vertical lines on the above image to complete the rectangular window surrounding the object in movement. This is implemented in CAPH with the `frame` actor.

6.2.3 Implementation

The CAPH implementation of this algorithm is given in listing 6.1. It consists of eight actors.

Listing 6.1: CAPH implementation of motion detection application

```
1  function abs x = if x < 0 then 0-x else x;
2  ------------------------
3  -- ACTORS
4  ------------------------
5  actor asub ()
6    ...
7  actor thr (k:unsigned<10>)
8    ...
9  actor hproj ()
10   ...
11 actor vproj ()
12   ...
13 actor area ()
14   ...
15 actor peak ()
```

```
16  ...
17 actor s1f ()
18  ...
19 actor frame ()
20 ...
21 ----------------------------
22 -- IOs
23 ----------------------------
24 stream i1:unsigned<8> dc from "camera:0";
25 ...
26 ------------------------------
27 -- Network declarations
28 ------------------------------
29 net diff = asub (i1,i2);
30 ...
31 net o = r;
```

– The `asub` actor computes the absolute value of the difference between two frames. Functionally speaking :

$asub : f_1\ f_2\ ...\ f_n,\ f_0\ f_1\ f_2\ ...\ f_{n-1} \longrightarrow f_1 - f_0\ f_2 - f_1\ ...\ f_n - f_{n-1}$

where f_i is a frame

and $f =<< p_{11}\ p_{12}\ ...\ p_{1n} >< p_{21}\ p_{22}\ ...\ p_{2n} > ... < p_{m1}\ p_{m2}\ ...\ p_{mn} >>$

The CAPH description of the actor is given in listing 6.2. The first input is the image coming from the camera and second is the image after a delay of one frame coming from memory. For the first image, it consists of zeros. In case of *control* tokens at inputs, the same token is written on output, for *data* tokens, it calculates the absolute of difference between pixels and writes result at output.

Listing 6.2: Absolute difference actor

```
1 actor asub ()
2   in  (a:unsigned<8> dc, b:unsigned<8> dc)
3   out (c:unsigned<10> dc)
4 rules (a,b) -> c
5 | ('<,'<) -> '<
6 | ('>,'>) -> '>
7 | ('p1,'p2) -> 'abs(p1-p2)
8 ;
```

– The `thr` actor writes 0 or 1 at output depending on whether the input value is greater than the threshold parameter `k` or not.

$thr(k) :< p_1\ p_2\ ...\ p_n > \longrightarrow < f(p_1)\ f(p_2)\ ...\ f(p_n) >$

where $f(p_i) = \begin{cases} 1\ if\ p_i{>}k \\ 0\ otherwise \end{cases}$

The CAPH description of this actor is given in listing 6.3. If input is *control* token then same token is written on output, if input is *data* token then 1 is produced at output if input is greater than `k` otherwise 0.

Listing 6.3: Threshold Actor

```
1 actor thr (k:unsigned<10>)
2   in  (a:unsigned<10> dc)
3   out (c:unsigned<1> dc)
4 rules a -> c
5 | '< -> '<
6 | '> -> '>
7 | 'p -> if p > k then '1 else '0
8 ;
```

– The `hproj` actor computes the horizontal projection of an image. This is done by adding all the values in a line. Functionally speaking :

$$
\begin{array}{lll}
hproj :< & & < \\
\quad < p_{11}\ p_{12}\ ...\ p_{1n} > & & \quad p_{11} + p_{12} + ... + p_{1n} \\
\quad < p_{21}\ p_{22}\ ...\ p_{2n} > & \longrightarrow & \quad p_{21} + p_{22} + ... + p_{2n} \\
\quad \vdots & & \quad \vdots \\
\quad < p_{m1}\ p_{m2}\ ...\ p_{mn} > & & \quad p_{m1} + p_{m2} + ... + p_{mn} \\
> & & >
\end{array}
$$

The CAPH description of this actor is given in listing 6.4. A local variable `s` is used to add pixels of a line. The first two rules correspond to the start and end of a frame respectively. In the third rule, the variable `s` is initialized from zero at the start of each line (line 9). The fifth rule (line 11) adds the value of each pixel to the existing value of `s`. In the fourth rule, the value stored in `s` is sent to the output at the end of a line (line 10).

Listing 6.4: Horizontal projection actor

```
1 actor hproj ()
2   in  (a:unsigned<1> dc)
3   out (c:unsigned<10> dc)
4 var s : unsigned<10>
5 var st : {S0,S1,S2} = S0
6 rules (st, a, s) -> (c, s, st)
7 |   (S0, '<, _) -> ('<, _, S1)
8 |   (S1, '>, _) -> ('>, _, S0)
9 |   (S1, '<, _) -> ( _, 0, S2)
10 |  (S2, '>, s) -> ('s, _, S1)
11 |  (S2, 'p, s) -> ( _, s+p, S2)
12 ;
```

– The `vproj` actor computes the vertical projection of an image. This is done by adding all

pixels of the same column. Functionally speaking :

$$
\begin{array}{lcl}
vproj :< & & < \\
\quad < p_{11}\ p_{12}\ ...\ p_{1n} > & & p_{11} + p_{21} + ... + p_{m1} \\
\quad < p_{21}\ p_{22}\ ...\ p_{2n} > & \longrightarrow & p_{12} + p_{22} + ... + p_{m2} \\
\quad \vdots & & \vdots \\
\quad < p_{m1}\ p_{m2}\ ...\ p_{mn} > & & p_{1n} + p_{2n} + ... + p_{mn} \\
> & & >
\end{array}
$$

The CAPH description of this actor is given in listing 6.5. To find the column-wise sum, each pixel value in a column is to be added until end of the image. This is implemented by using an array `z`. Each element of the array correspond to a column in the image. The first rule initializes the array to zero at the start of each frame (line 9). In the fourth rule (line 12), each pixel is added to the previous stored value at the corresponding index of the array and the result is stored at the same index of the array. At the end of a frame, the last rule (lines 14-16) sends the values stored in the array to the output and starts processing the next frame by moving to the first rule.

Listing 6.5: Vertical projection actor

```
1  actor vproj ()
2    in  (a:unsigned<1> dc)
3    out (c:unsigned<10> dc)
4  var s : {S0,S1,S2,S3}=S0
5  var z : unsigned<10> array[512] = [ 0 : 512 ]
6  var i : unsigned<10>
7  var w : unsigned<10>
8  rules (s,a,z, i, w) -> (s, c, z, i, w)
9  | (S0, '<, z, _, _) -> (S1, _, z[i in 0..511 <- 0] , _, _)
10 | (S1, '>, z, i, _) -> (S3,'<, _, 0, i)
11 | (S1, '<, _, _, _) -> (S2, _, _, 0, _)
12 | (S2, 'p, z, i, _) -> (S2, _, z[i<-z[i]+p], i+1, _)
13 | (S2, '>, _, _, _) -> (S1, _, _, _, _)
14 | (S3, _, z, i, w) -> (if i<w then S3 else S0 ,
15                          if i<w then 'z[i] else '>,
16                          _ ,i+1 , _ )
17 ;
```

– The `area` actor analyses the thresholded horizontal/vertical projection and gives the whole area where a moving object can be found. Functionally speaking :

$$area :< p_1\ p_2\ ...\ p_n > \longrightarrow < f(p_1)\ f(p_2)\ ...\ f(p_n) >$$

$$where\ f(p_i) = \begin{cases} 1\ if\ k < i < l\ (k\ is\ start\ of\ area\ and\ l\ is\ end) \\ 0\ otherwise \end{cases}$$

The CAPH description of this actor is given in listing 6.6. It consists of two steps. In the first step, indexes of the boundary of the area are calculated. This is accomplished in the third and sixth rules (lines 12 and 15) and results are stored in the `min` and `max` variables. In the second step, the output is written according to the boundary indexes.

This is accomplished in the last rule (lines 17-24), where the value sent to the output is 1 for all indices between `min` and `max` and 0 for others.

Listing 6.6: Area computation actor

```
1  actor area ()
2    in  (a:unsigned<1> dc)
3    out (c:unsigned<1> dc)
4  var s : {S0,S1,S2,S3}=S0
5  var min : unsigned<10>
6  var max : unsigned<10>
7  var w : unsigned<10>
8  var i : unsigned<10>
9  rules (s,a, i,min,max, w) -> ( s, c, i,min,max, w)
10 | (S0, '<,_ , _, _, _) -> (S1, _, 0, 0, 0, _)
11 | (S1, '0, i, _, _, _) -> (S1, _,i+1, _, _, _)
12 | (S1, '1, i, _, _, _) -> (S2, _,i+1, i, _, _)
13 | (S1, '>, i, _, _, _) -> (S3,'<, 0, _, _, i)
14 | (S2, '0, i, _, _, _) -> (S2, _,i+1, _, _, _)
15 | (S2, '1, i, _, _, _) -> (S2, _,i+1, _, i, _)
16 | (S2, '>, i, _, _, _) -> (S3,'<, 0, _, _, i)
17 | (S3,_,i,min,max,w) ->
18     (if i<w then S3 else S0,
19      if i<w then
20        if(i>min && i<max)
21          then '1
22        else '0
23      else '>,
24      i+1, _, _, _ );
```

– The `peak` actor analyses the thresholded horizontal/vertical projection and gives only boundaries of the area where moving object can be found. Functionally speaking :

$$peak :< p_1\ p_2\ ...\ p_n > \longrightarrow < f(p_1)\ f(p_2)\ ...\ f(p_n) >$$

$$where\ f(p_i) = \begin{cases} 1\ if\ (k < i < k+10)\ or\ (l-10 < i < l)\ (k\ is\ start\ of\ area\ and\ l\ is\ end) \\ 0\ otherwise \end{cases}$$

The CAPH description of this actor is given in listing 6.7. The difference from the previous actor (i.e.`area`) is in the last rule in which the result is written to the output. Here instead of sending the 1s for the whole area, 1s are sent only for the boundary of the area. The boundary width is set here to `10`, which can also be changed.

Listing 6.7: Peak computation actor

```
1  actor peak ()
2    in  (a:unsigned<1> dc)
3    out (c:unsigned<1> dc)
4  var s : {S0,S1,S2,S3}=S0
5  var min : unsigned<10>
6  var max : unsigned<10>
7  var w : unsigned<10>
8  var i : unsigned<10>
9  rules (s, a, i, min, max, w) -> (s, c, i, min, max, w)
10 | (S0, '<, _, _, _, _) -> (S1, _, 0, 0, 0, _)
11 | (S1, '0, i, _, _, _) -> (S1, _, i+1, _, _, _)
12 | (S1, '1, i, _, _, _) -> (S2, _, i+1, i, _, _)
```

```
13 | (S1, '>, i, _, _, _) -> (S3, '<, 0, _, _, i)
14 | (S2, '0, i, _, _, _) -> (S2, _, i+1, _, _, _)
15 | (S2, '1, i, _, _, _) -> (S2, _, i+1, _, i, _)
16 | (S2, '>, i, _, _, _) -> (S3, '<, 0, _, _, i)
17 | (S3, _,i, min, max, w) ->
18     (if i<w then S3 else S0 ,
19      if i<w then
20        if((i>min && i<min+10 )||(i<max && i>max-10 ))
21          then '1
22          else '0
23      else '>,
24      i+1 , _ , _ , _ )
25 ;
```

– `s1f` performs a skip frame operation on the first image frame. The purpose of this actor is to compensate for a FIFO equal to the size of one image frame, which results in reducing the number of memory bits utilized on FPGA. Functionally speaking :

$s1f : f_1\ f_2\ ...\ f_n \longrightarrow f_2\ f_3\ ...\ f_n$

where f_i is a frame

and $f =<< p_{11}\ p_{12}\ ...\ p_{1n} >< p_{21}\ p_{22}\ ...\ p_{2n} > ... < p_{m1}\ p_{m2}\ ...\ p_{mn} >>$

The CAPH description of this actor is given in listing 6.8. For the first frame, input is consumed without writing to the output. The first five rules (lines 6-10) perform this task. The next five rules (lines 11-15), read the input value and write the same value to the output which will continue for all the incoming frames.

Listing 6.8: Skip one frame actor

```
1  actor s1f ()
2    in  (a:unsigned<8> dc)
3    out (c:unsigned<8> dc)
4  var s : {S0,S1,S2,S3,S4,S5} = S0
5  rules (s,a) -> (s,c)
6  | (S0,'<) -> (S1,_)
7  | (S1,'>) -> (S3,_)
8  | (S1,'<) -> (S2,_)
9  | (S2, _) -> (S2,_)
10 | (S2,'>) -> (S1,_)
11 | (S3,'<) -> (S4,'<)
12 | (S4,'>) -> (S3,'>)
13 | (S4,'<) -> (S5,'<)
14 | (S5,'p) -> (S5,'p)
15 | (S5,'>) -> (S4,'>)
16 ;
```

– The `frame` actor draws a window around the detected object. Functionally speaking :

$$
\begin{array}{l}
frame :< q_1\ q_2\ ...\ q_n >, \\
\quad < r_1\ r_2\ ...\ r_m >, \\
\quad < \\
\quad\ < p_{11}\ p_{12}\ ...\ p_{1n} > \\
\quad\ < p_{21}\ p_{22}\ ...\ p_{2n} > \\
\qquad\qquad \vdots \\
\quad\ < p_{m1}\ p_{m2}\ ...\ p_{mn} > \\
\quad >
\end{array}
\longrightarrow
\begin{array}{l}
< \\
\quad < f(q_1, r_1, p_{11})\ f(q_2, r_1, p_{12})\ ...\ f(q_n, r_1, p_{1n}) > \\
\quad < f(q_1, r_2, p_{21})\ f(q_2, r_2, p_{22})\ ...\ f(q_n, r_2, p_{2n}) > \\
\qquad\qquad\qquad\qquad \vdots \\
\quad < f(q_n, r_m, p_{m1})\ f(q_n, r_m, p_{2n})\ ...\ f(q_n, r_m, p_{mn}) > \\
>
\end{array}
$$

where $f(q, r, P) = \begin{cases} 1\ if\ q = 1\ and\ r = 1 \\ P\ otherwise \end{cases}$

The CAPH description of this actor is given in listing 6.9. It uses the output produced by `peak` and `area` actors to draw a window on the image. In fact, two `frame` actors are used, first to draw the horizontal lines of the window and second to draw the vertical lines of the window.

The input to the first `frame` actor are `peak` of horizontal projection, `area` of vertical projection and image obtained by `s1f` actor. The objective is to draw horizontal lines at the boundary of the horizontal projection. The area of the vertical projection is used to restrict the width of lines in the area where the vertical projection is also found.

The input to the second `frame` actor is `peak` of vertical projection, `area` of horizontal projection and output of the first `frame` actor. This actor will draw vertical lines at the boundary of the vertical projection to complete the window. The area of the horizontal projection is used to restrict the length of lines in the area where the horizontal projection is also found.

The output of the `frame` actor is `1` when both first inputs are 1, otherwise the same pixel read from third input will be sent to the output. In both instances of the `frame` actor, the vertical projection (either area or peak) is the first input. This is saved in an array `z` during the processing of the first line to be used for later lines (lines 13/14 and 19). On the other hand, the horizontal projection (either area or peak) is the second input and is read at the start of each line (lines 12 and 18).

Listing 6.9: Windows drawing actor

```
1  actor frame ()
2    in  (a:unsigned<1> dc, b:unsigned<1> dc,c:unsigned<8> dc)
3    out (d:unsigned<8> dc)
4  var s : {S0,S1,S2,S3,S4,S5,S6}=S0
5  var z : unsigned<1> array[512] = [ 0 : 512 ]
6  var y : unsigned<1>
7  var i : unsigned<10>
8  rules (s, a, b, c, y, z, i) -> (s, d, y, z, i)
9  | (S0, _, _,'<, _, _, _) -> (S1, '<, _, _, _)
10 | (S1, _, _,'>, _, _, _) -> (S0, '>, _, _, _)
```

```
| (S1,'<,'<,'<, _, _, _) -> (S2, '<, _, _, 0)
| (S2, _,'t, _, _, _, _) -> (S3, _, t, _, _)
| (S3,'1, _,'p, y, z, i) -> (S3,if y=1 then '1 else 'p, _,z[i<-1], i+1)
| (S3,'0, _,'p, _, z, i) -> (S3, 'p, _, z[i<-0], i+1)
| (S3,'>,'>, _, _, _, _) -> (S4, '>, _, _, _)
| (S4, _,'>,'>, _, _, _) -> (S0, '>, _, _, _)
| (S4, _, _,'<, _, _, _) -> (S5, '<, _, _, 0)
| (S5, _,'t, _, _, _, _) -> (S6, _, t, _, _)
| (S6, _, _,'p, y, z, i) -> (S6,if(z[i]=1 && y=1)then '1 else 'p, _, _, i+1)
| (S6, _, _,'>, _, _, _) -> (S4, '>, _, _, _)
;
```

The I/O streams are declared in listing 6.28. The first input is the image obtained from the camera and second is the image obtained from memory after a delay of one frame in the input image. This delay is obtained by generating zeros for the first image and at the same time saving the image in the SDRAM. For the next image frames, the image coming from the camera is saved in the memory and the existing image is sent to the CAPH program.

Listing 6.10: I/O Streams

```
stream i1:unsigned<8> dc from "camera:0";
stream i2:unsigned<8> dc from "mem:0";
stream o:unsigned<8> dc to "display:0";
```

The network declaration is defined in listing 6.11 which will generate the dataflow graph shown in figure 6.8.

Listing 6.11: Network declaration

```
net diff = asub (i1,i2);
net thr_d = thr [80] diff;
net hp = hproj thr_d;
net hp_t = thr [50] hp;
net ah = area hp_t;
net pa = peak hp_t;
net sf = s1f i1;
net vp = vproj thr_d;
net vp_t = thr [50] vp;
net av = area vp_t;
net pv = peak vp_t;
net vf = frame (av,ph,sf);
net r  = frame (pv,ah,vf);
net o = r;
```

The results of the experiment performed on the SeeMOS platfrom are shown in figure 6.9. Figure 6.9a and 6.9b are the two consecutive frames of the input image. The thresholded difference of these two images is given in figure 6.9c. Horizontal and vertical projections of the thresholded image are given in figures 6.9d and 6.9e respectively. The final result is shown in figure 6.9f where a rectangular window is drawn around the moving object.

6.2.4 Performance Results

The application processes on the fly on video streams of 512 x 512 x 8 bit images at 15 FPS. The performance results are summarized in table 6.2. The application achieves a maximum

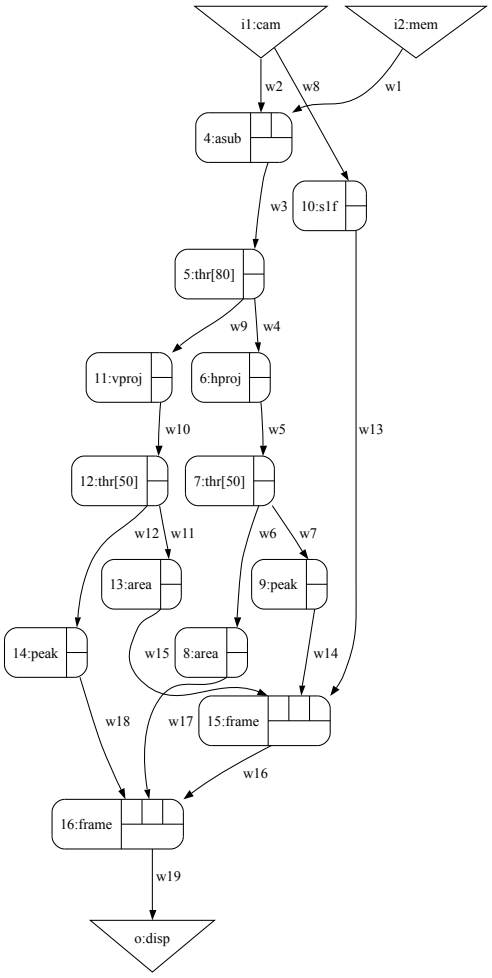
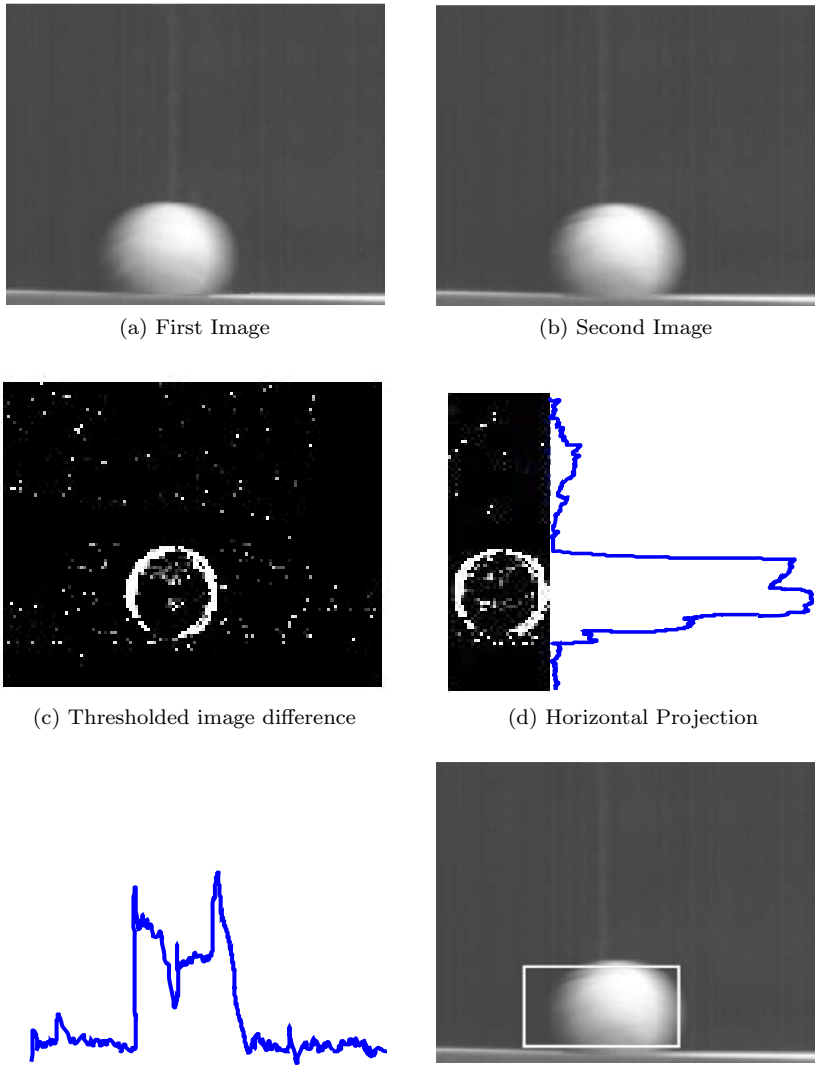


Figure 6.8: Dataflow graph of motion detection application

clock frequency of 150 MHz. It uses 3550 logic elements (6%), 17 kbits of memory bits (<1% of total) and 512 kB of external meory. The external memory is used to store one image frame in order to create a delay of one frame.

Table 6.2: Motion Detection Application Performance Results

	Total	Used
Max Frequency		150 MHz
Logic Elements	57120	3,550 (6%)
Memory Bits	5,215,104	17 kbits (<1%)
SRAM Blocks	5	1 (20%)
DSP Blocks	144	0 (0%)



(a) First Image

(b) Second Image

(c) Thresholded image difference

(d) Horizontal Projection

(e) Vertical Projection

(f) Final result

Figure 6.9: Motion detection application results

6.3 Connected Component Labeling

6.3.1 Objective

Connected component labeling (CCL) is used in many image processing applications, especially in region segmentation algorithms [87]. The task is to assign a unique and different label to each connected component of an object in an image. This is usually accomplished in several steps. First thresholding is applied to the image to differentiate objects from the background. Connected component labeling is then applied to this preprocessed binary image, to assign a unique label to each object. Finally, each object is processed (for example to assign a different color to each object). Many algorithms have been proposed to implement this [88, 89, 90, 91, 92, 93, 94, 95, 96]. The next section will describe the basic principle adopted by all the algorithms, the reason for their difference and finally the approach adopted for CAPH implementation.

6.3.2 Principle

The classical connected component labeling algorithm [97] requires two passes through the image. In the first pass, each pixel of the image is assigned a label according to the following rules :

- If the pixel is a background pixel, it is assigned the label zero.
- If the pixel is an object pixel, 2 neighbors (in case of 4-connectivity as shown in figure 6.10) are examined.
  - If both neighbors are background pixels then a new label is assigned to the current pixel.
  - If both neighbors have the same label, this label is also assigned to the current pixel.
  - If both neighbors have different labels then this indicates a merging condition.

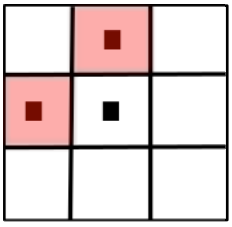


Figure 6.10: 4-Pixel Connectivity

In case of a merging condition, one of the two labels will continue to be used and all instances of the other label have to be replaced with the label retained. For example, let us dconsider a U shaped object as shown in figure 6.11. Because pixels are generally read in raster row-major order, there is no prior information that pixel **p1** and **p2** belong to the same object until pixel **p3** is reached. The main difficulty with the merging step is to relabel all the previous pixels belonging to the object. This requires one more scan through the object. Since an image may contain many merging cases, the relabeling process is carried out at the end of the image. A merging table is maintained to keep a record of all the label equivalences introduced by

the merging cases. In the second pass, the equivalence table is used to relabel all the pixels in the image. Based on this approach, several different ways of handling equivalences have been proposed in the literature [97, 98, 99]. They differ in the data structure used to store the equivalences as well as in the method used to merge objects by these equivalence [100]. This problem becomes more important for resource constraint devices like FPGAs. There are number of variations of this algorithm proposed for FPGA implementation [101] : two pass (classical) [97], multiple scan [102], parallel processing [103], contour tracing [104] and single pass [105].

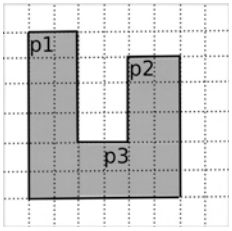
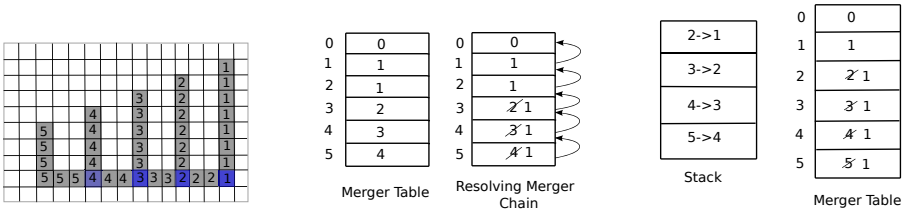


Figure 6.11: Label merging in U-Shaped Object

Single pass algorithm was developed for streaming data systems. Normally, the objective of connected component labeling is to extract features from each region, where labeling is used to separate regions. In these scenarios, there is no need to label the image provided that features can be correctly extracted for each region. So, these algorithms extract data to calculate features during the first pass. There is no need to buffer input image frames which results in low memory requirements. However, single pass algorithms do have some limitations compared to two pass algorithms. These are good for object counting and features of interest (position, size, area etc.) but can not be used where a labeled object mask is required. Since our objective is to assign different colors to objects to uniquely identify them, which means labeling objects rather than feature extraction, a second pass through the image frame is required.

The merging step is implemented according to [106]. Whenever a merging situation occurs, two tasks are performed. First, the smaller of the two labels is assigned to the current pixel and secondly, the bigger label points to the smaller label in the equivalence/merging table. Merging table is used as a look-up table on the output of the row buffer[2]. This ensures the assignment of correct labels to previously stored pixels which have been subsequently merged. The merging situation is complicated when there are multiple merging cases on the same line. The selection of the smaller of the two labels solves the problem when the smaller label is on the left, as all instances of the larger label are changed to the smaller one. But a problem arises when during a merger the smaller label is on the right. This scenario is called merger chaining. This chaining should be resolved before the start of the next row. Consider the example in figure 6.12a, where blue boxes represent merging labels. At the end of the row, the merger table contains incorrect final labels for 5,4 and 3 as shown in the left table in figure 6.12b. One simple approach can be to search the whole merger table at the end of each line to resolve this chain as shown in the right table in figure 6.12b. But this can take significant time when this table is big[3]. The other way is to record new mergers for each row. Here only mergers with smaller labels on the right are recorded. The relevant mergers are pushed onto a stack and then popped off at the end of line to resolve the mergers. This is accomplished in three steps as shown in figure 6.12c:

[2]used to save the labels for the previous row
[3]The worst case for an MxN image can be (M/2)*(N/2)



(a) Merger Chain (b) Merger Table (c) Resolution using stack

Figure 6.12: Merger chain and its resolution

1. The larger of the merged pair and the merger target (smaller of the pair) are popped from the stack.
2. The target is looked up in the merger table to obtain the final target.
3. If the final target is different from that popped of the stack, this target is saved for the larger of the merger table in the merger table.

The steps of the algorithm implemented as shown in figure 6.13, are :

Thresholding the input image to remove noise and obtain a binary image. This is implemented in CAPH with the `thr` actor.

Assigning different labels to each connected object in this binary image. This is implemented in CAPH with the `ccl` actor.

Relabeling the object labels in the equivalence table to reduce the label count as many temporary labels are used. This is implemented in CAPH with the `resLabel` actor.

Resolving the different object labels assigned to one object by reassigning labels from equivalence table. This is implemented in CAPH with the `resLabel` actor.

Assigning different color to each object. This is implemented in a C++ application displaying final image on the screen.

6.3.3 Implementation

The CAPH code for the application is given in listing 6.12. It consist of three *actors*.

Listing 6.12: CAPH implementation of CCL application

```
function lbl(p, d, b, l) =
  if p=0 then 0
  else if b=0 && d=0 then l
  else if b=d then b
  else if d=0 then b
  else if b=0 then d
  else if b<d then b
  else d
  : unsigned<1> * unsigned<8> * unsigned<8> * unsigned<8> > unsigned<8>;

   ACTORS

actor ccl ()
```

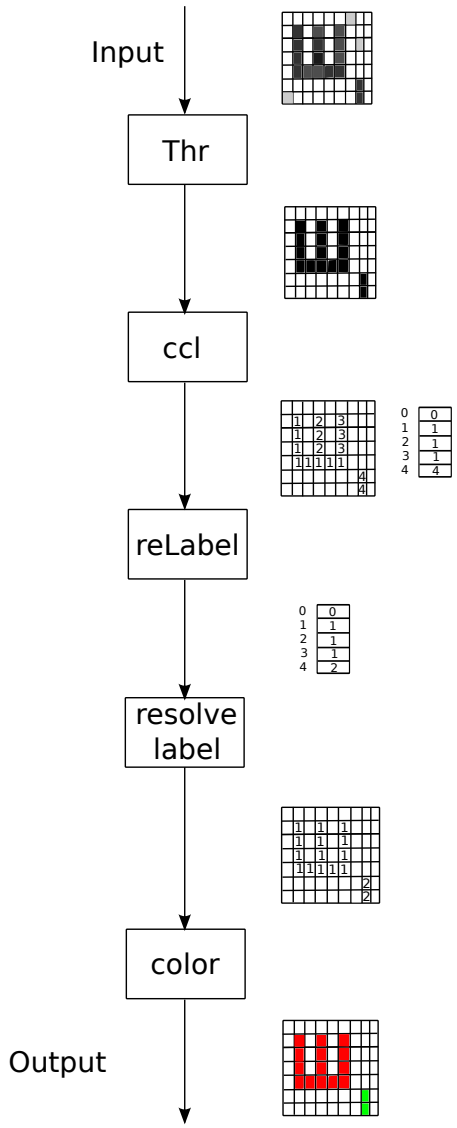


Figure 6.13: Different steps of CCL application

```
14 ...
15 actor resLabel()
16 ...
17 actor thr (k:unsigned<8>)
18 ...
19 ------------------------
20 -- IOs
21 ------------------------
22 stream i:unsigned<8> dc from "camera:0";
23 stream o:unsigned<8> dc to "display";
24 ------------------------
25 -- Network declarations
26 ------------------------
27 net (a,b) = (ccl (thr [4] i));
28 net o = resLabel(a,b);
```

– The `thr` actor is used to remove noise and obtain a binary image. Its implementation is

same as in the previous application in section 6.2.

– The `ccl` actor implements connected component labeling algorithm on the above binary image. The code for this actor is given in listing 6.13. It performs *object labeling*, *merging* and *merger chaining* to label the objects in the image. Depending on the input image pixel and two neighborhood pixel values, any of these tasks are performed. These are implemented using *guards* on *transition rules* in lines 20-33. Of the two neighborhood pixels, one is the previous pixel in variable `d` (`d1p` described in section 5.2) and other is the previous line pixel in array variable `b` (`d1l` described in section 5.2).

  The rule corresponding to *merging* is given in lines 20-22. It executes when both the neighborhood pixels are neither zero, nor equal to each other and the previous line pixel is greater than the previous pixel as given in the `guards` condition. The action part of this rule updates the merged labels in the equivalence table `et` (declared in line 8). Since, smaller of the labels is used, the entry for bigger label in equivalence table is updated with smaller label (by assigning the value of `d` at the index of `b[i]`). Along with this, the current pixel is assigned the value of `d` (i.e. smaller of the two labels).

  The rule corresponding to *merger chaining* is given in lines 23-25. This rule executes when the previous line pixel is less than the previous pixel in the *guards* condition. Upon execution, this rule keeps the mergers in a stack instead of directly updating the merger table. This stack saves both neighborhood pixel labels. This is implemented in CAPH by using two arrays `st1` and `st2` (declared in lines 9 and 10). The array index variable `si` keeps track of the number of merger chains occurring during each row. Apart from recording merger chains, this rule also assigns the smaller label to the current pixel (i.e. value of the previous line pixel from array `b`). At the end of each row, the merger chains are resolved and the merger table is updated accordingly by moving to the ninth rule in 36-37. After completing the merger chain resolution, the stack index `si` is reinitialized to zero and control is moved to the 3rd rule to start processing the next line.

  All of the other conditions are handled by the rule in lines 26-33 which contain no guard conditions. The conditions in lines 27-31 decide the label for the current pixel based on the value of the current pixel and the labels of two the neighborhood pixels. There is also a condition to assign a new label to the current pixel. The variable `l` is used for the label counter. When the current pixel is assigned a new label, the label counter `l` is incremented to keep the count updated (in line 32).

  It is also important to note that arrays containing the previous line and the equivalence table are reinitialized at the start of each frame, otherwise they will contain values from the last line of the previous frame which will result in invalid label assignment for the objects in the new frame. This is done in the first rule in lines 16-17.

  This actor consists of two outputs. The first is the image containing the labeled pixels, sent to the output after reading the input pixel and executing the corresponding rule. The second is the equivalence table which is sent at the end of each frame. So, at the end of a frame, the last rule in line 40 is executed to send the equivalence table to the output before starting processing the next frame.

Listing 6.13: CCL actor

```
actor ccl ()
  in  (a:unsigned<1> dc)
  out (c:unsigned<8> dc,e:unsigned<8> dc)
var s : {S0,S1,S2,S3,S4,S5,S6}=S0
var d : unsigned<8>
var bt : unsigned<8>
var b : unsigned<8> array[256] = [ 0 : 256 ]
var et : unsigned<8> array[200] = [i in 0..199 <- i]
var st1 : unsigned<8> array[50] = [i in 0..49 <- i]
var st2 : unsigned<8> array[50] = [i in 0..49 <- i]
var si : unsigned<8>
var i : unsigned<8>
var l : unsigned<8> = 2
rules
   ( s, a, b, d, i, l, et, st1, st2, si) -> ( s, c, e, l, et, b, d, i, st1, st2, si)
| (S0,'<, b, _, et, _, _, _, _) -> (S1,'<,_,2,et[i in 0..199<-i],
                                        b[i in 0..255<-0], _, _, _, _, _)
| (S1,'>, _, _, _, _, _, _, _) -> (S5,'>, _, _, _, _, _, _, _, _)
| (S1,'<, _, _, _, _, _, _, _) -> (S2,'<, _, _, _, _, 0, 0, _, _,1)
| (S2,'p, b, d, i, l, et, st1, st2, si)when  b[i]!=0 && d!=0 && d!=b[i] && b[i]>d->
    (S2, if p=0 then '0 else 'd ,_,l, et[b[i]<-d], b[i<-et[lbl(p, d, b[i], l)]],
         et[lbl(p,d,b[i],l)], i+1, st1, st2, si)
| (S2,'p, b, d, i, l, et, st1, st2, si)when  b[i]!=0 && d!=0 && d!=b[i] && b[i]<d->
    (S2, if p=0 then '0 else 'b[i], _, l, et, b[i<-et[lbl(p,d,b[i],l)]],
         et[lbl(p,d,b[i],l)], i+1, st1[si<-d] ,st2[si<-b[i]], si+1)
| (S2, 'p, b, d, i, l, et, st1, st2, si) ->
   (S2, if p=0 then '0
         else if  b[i]=0 && d=0  then 'l
         else if b[i]=d then 'b[i]
         else if d=0 then 'b[i]
         else 'd,
         _,if b[i]=0 && d=0 && p!=0 then l+1 else l,
         et, b[i<-et[lbl(p,d,b[i],l)]],et[lbl(p,d,b[i],l)],i+1, st1, st2, si)
| (S2, >, _, _, _, _, _, _, _, si) -> (S3,'>, _, _, _, _, _,_,_,_,si-1)
| (S3, _, _, _, _, _, et, st1, st2, 0) -> (S1, _, _, _, _, _, _,_,_,_, _)
| (S3, _, _, _, _, _, et, st1, st2, si) -> (S3, _, _, _,
                                        et[st1[si]<-et[st2[si]]],_,_,_,_,_,_,si-1)
| (S5, _, _, _, _, _, _, _, _, _) -> (S6, _,'<, _, _, _, _, 0,_,_, _)
| (S6, _, _, _,200, _, _, _, _, _) -> (S0, _,'>, _, _, _, _, 0,_,_, _)
| (S6, _, _, _, i, _, et, _, _, _) -> (S6, _,'et[i],_,_,_,_,i+1,_,_, _)
;
```

– The `resLabel` actor given in listing 6.14 is used to resolve equivalence labels. It receives the equivalence table from the `ccl` actor and relabels the objects according to this table. This requires one FIFO (FIFO 8 in figure 6.14) equal to the size of one frame as the `ccl` actor sends this table at the end of a frame. Apart from resolving labels, another task performed by this actor is the relabeling of object labels in the equivalence table. Since many temporary labels are assigned to an object, this increases the label count. It is possible that the first object has label $\mathbf{2}^4$ and the second object has a label **100**. This creates a problem when assigning different colors to each object. So, before relabeling the objects of an image, the equivalence table is organized to give labels to each object without wasting temporary used labels. This is achieved by having two arrays of equivalence labels

[4] label counter is initialized from 2

: `et` and `et1` (declared in lines 5 and 6). The first is used to create an updated equivalence table and the second to point to the old labels to the new ones.

The relabeling is implemented in CAPH by using two rules in lines 13-15. The first one contains a `guard` condition that compares the input value `v` with the previous input value `z`. In case these are not equivalent and the `et1` array for the current label points to zero, a new label `lb` is assigned to the current label. The `et` array assigns this new label to the current index and the `et1` array will point the old label to the new one (line 14). In case the `guard` condition is not true, the rule in line 15 is executed. It reads the value pointed at the `et1` array for the current label and saves it in the current index of the `et` array.

After organizing the labels in the equivalence table, the input image from the first input is read and objects are relabeled according to the updated equivalence table. This is done by reading the input label for the pixel and sending to the output the corresponding label stored in the `et` array (line 21).

Both the arrays used to store equivalence tables are reinitialized at the start of each frame to avoid any confusion with the old values. This is achieved by the initialization statements in the first rule (line 12).

Listing 6.14: Resolve label actor

```
1  actor resLabel()
2    in  (a:unsigned<8> dc, b:unsigned<8>dc)
3    out (c:unsigned<8> dc)
4  var s : {S0,S1,S2,S3,S4} = S0
5  var et : unsigned<8> array[200] = [i in 0..199 <- i]
6  var et1 : unsigned<8> array[200] = [0 : 200]
7  var z : unsigned<8>
8  var lb : unsigned<8> = 2
9  var i : unsigned<8>
10 rules
11   (s, a, b, et, et1, i, z, lb) -> ( s, c, et, et1, i, z, lb)
12 |(S0,_,'<,et,et1,_,_,_,_)->(S1,_,et[i in 0..199<-i],et1[i in 0..199<-0],0,0,_)
13 | (S1, _, 'v, et, et1, i, z, lb) when v>z && et1[v]=0 && v>1  ->
14       (S1, _,et[i<-lb],et1[v<-lb], i+1, v, lb+1)
15 | (S1, _, 'v, et,et1, i, z, _) -> (S1, _,et[i<-et1[v]], _, i+1, v, _)
16 | (S1, _, '>, _, _, _, _, _) -> (S2, _, _, _, 0, _, _)
17 | (S2, '<, _, _, _, _, _, _) -> (S3, '<, _, _, _, _, _)
18 | (S3, '<, _, _, _, _, _, _) -> (S4, '<, _, _, _, _, _)
19 | (S3, '>, _, _, _, _, _, _) -> (S0, '>, _, _, _, _, _)
20 | (S4, '>, _, _, _, _, _, _) -> (S3, '>, _, _, _, _, _)
21 | (S4, 'v1, _, et, _, _, _, _) -> (S4,'et[v1], _, _, _, _, _)
22 ;
```

The dataflow graph of the application is shown in figure 6.14. The results of the experiment are shown in figure 6.15. Figure 6.15a is the input image and figure 6.15b is the image obtained after thresholding the input image. The final result with each object assigned a different color is shown in figure 6.15c. The RTL view of the CCL application is given in figure 6.16. The assignment of logic elements (LEs) and memory blocks is shown in figure 6.17, where light blue color represents logic elements unused and dark blue color is for logic elements used. Similarly, two dark green boxes represent the MRAM blocks used.

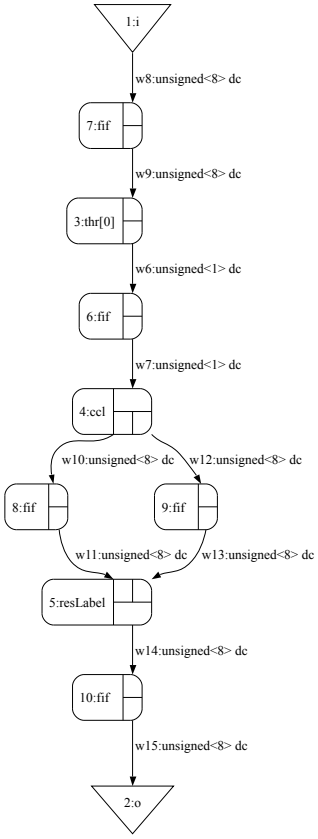


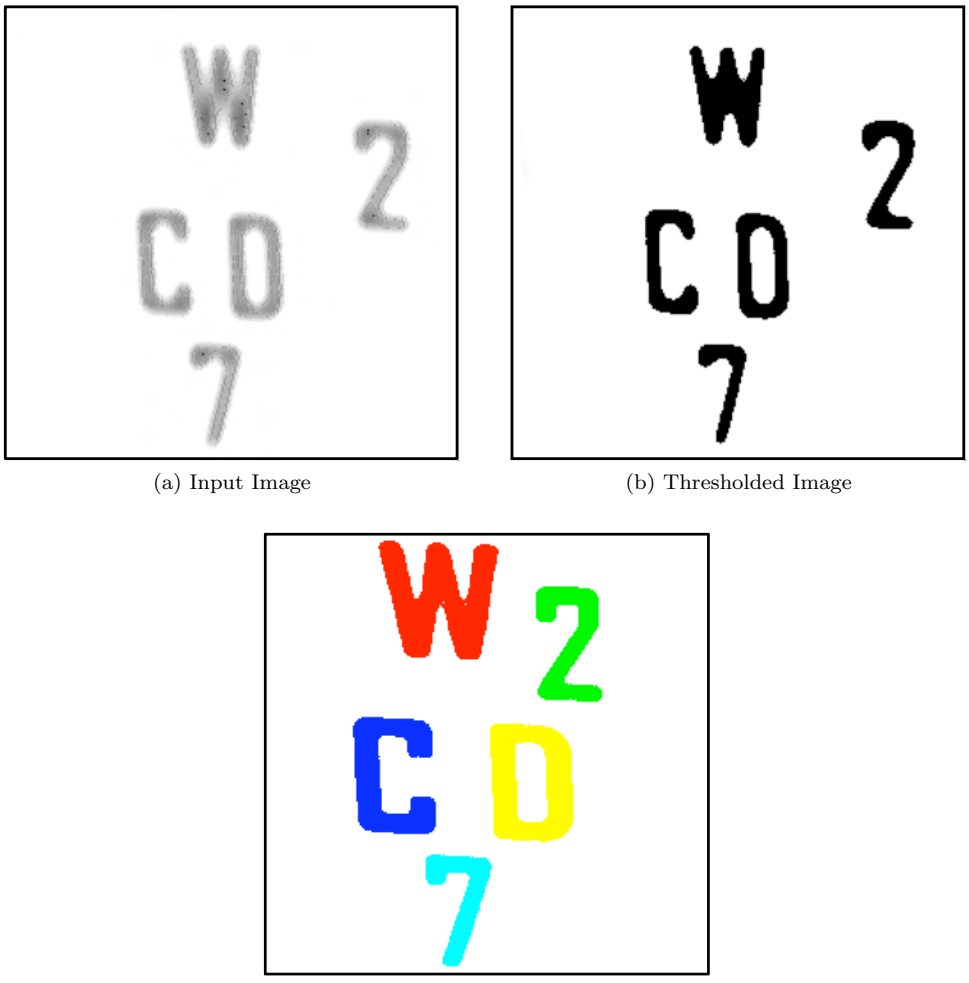
Figure 6.14: Dataflow Graph of CCL application

6.3.4 Experimental Results

The application processes on the fly video streams of 256 x 256 x 8 bit images at 20 FPS. The performance results are summarized in table 6.3. The application achieves a maximum clock frequency of 30 MHz. It uses 16559 logic elements (29%), 665360 memory bits (<13% of total). The memory bit consumption is large in this experiment because of the size of one FIFO which is equal to one image frame.

Table 6.3: CCL Application Performance Results

	Total	Used
Max Frequency		30 MHz
Logic Elements	57120	16,559 (29%)
Memory Bits	5,215,104	665360 (<13%)
DSP Blocks	144	0 (0%)



(a) Input Image

(b) Thresholded Image

(c) Final result

Figure 6.15: CCL application results

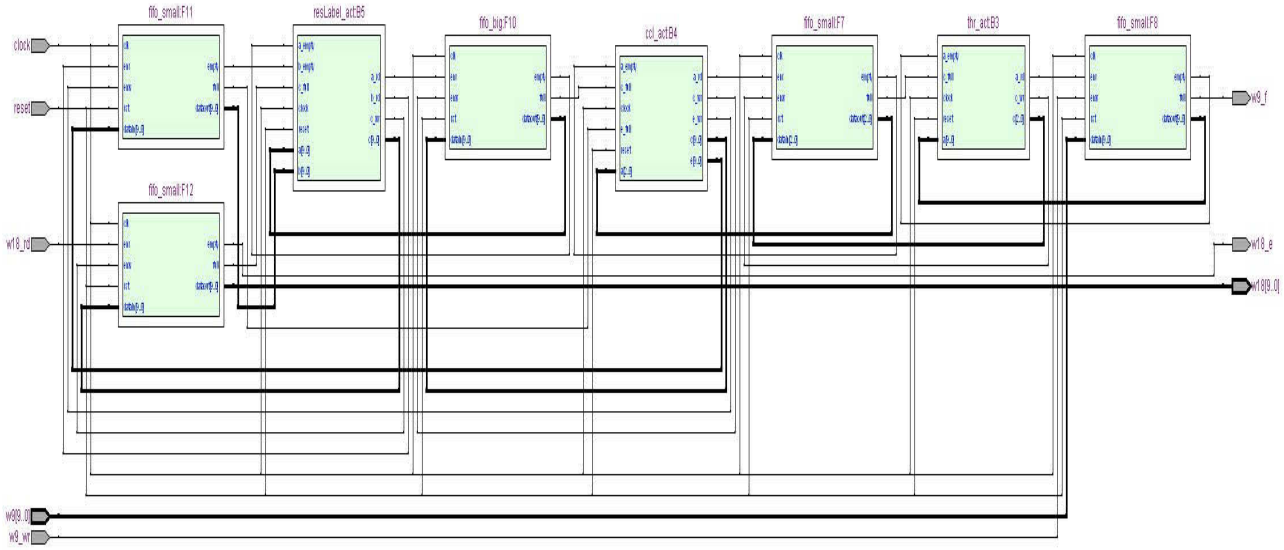


Figure 6.16: RTL view of CCL application

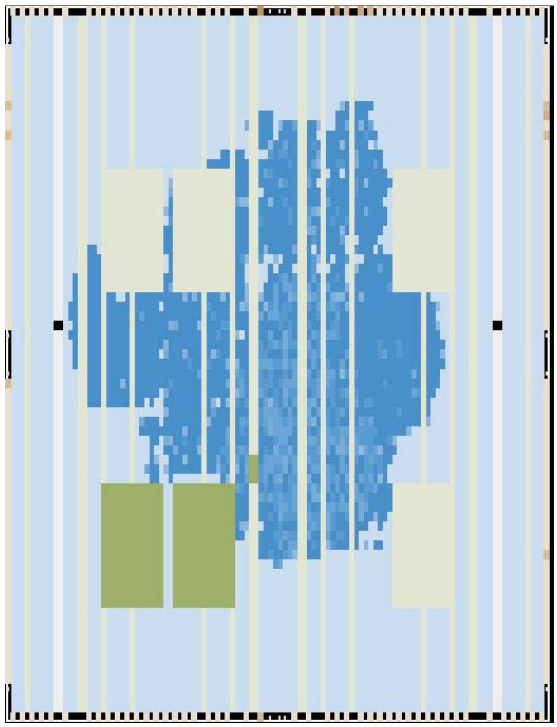


Figure 6.17: FPGA floorplan of CCL application

6.4 JPEG Encoder

6.4.1 Objective

JPEG also known as ISO/IEC IS 10918-1 or ITU-T Recommendation T.81, is widely used standard for image compression. *Recommendation T.81* [107] is a document published by the international standards bodies ITU-T (International Telecommunication Union) and ISO/IEC (International Organization for Standardization / International Electrotechnical Commission). According to this document, JPEG consists of several blocks such as Motion Estimation and Compensation (ME/MC), Discrete Cosine Transformation (DCT), Quantization, ZigZag scan, Run Length Encoding (RLE) and Variable Length Coding (VLC). The next section will describe the parts of JPEG used for implementation i.e. DCT, Quantization, Zigzag scan and RLE.

6.4.2 Principle

6.4.2.1 Discrete Cosine Transformation (DCT)

The discrete cosine transform (DCT) [108] has been widely applied to many image and video compression standards such as JPEG, MPEG and H.264 in order to reduce the spatial redundancies in the correlation of signals. It transforms a signal or image from the spatial domain to the frequency domain. It has the property that, for a typical image, most of the visually significant information about the image is concentrated in just a few coefficients of the DCT. DCT is block based transformation. The size of block is not fixed but all the standards, including JPEG and MPEG/H.264 use an 8x8 block size.

Mathematically, the 1D DCT of a sequence of length N is described as

$$(\mathbf{X})_u = \frac{C(u)}{\sqrt{2N}} \sum_{i=0}^{N-1} (\mathbf{X})_i \cos \frac{(2i+1)u\pi}{2N}, \tag{6.1}$$

where $0 \leq u < N-1$ and

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}} & u = 0 \\ 1 & u > 0 \end{cases}$$

Here $(X)_i$ is input and $(X)_u$ is transformed output. We are concerned with the implementation of DCT for images. Since images are two-dimensional, this necessitates the extension of DCT to a two-dimensional space [109, 110]. The 2-D DCT is a direct extension of the 1-D and is given by

$$(\mathbf{X})_{u,v} = \frac{C(u)}{\sqrt{2N}} \frac{C(v)}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (\mathbf{X})_{i,j} \cos \frac{(2i+1)u\pi}{2N} \cos \frac{(2j+1)v\pi}{2N}, \tag{6.2}$$

where $0 \leq u, v < N-1$ and

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}} & u = 0 \\ 1 & u > 0 \end{cases}$$

and

$$C(v) = \begin{cases} \frac{1}{\sqrt{2}} & v = 0 \\ 1 & v > 0 \end{cases}$$

$(X)_{i,j}$ is the pixel at coordinate (i, j) in the image. This equation calculates one value of a

transformed image from one pixel of the original image. N is the size of block to which DCT is applied. By replacing N with 8 as explained earlier, i and j range from 0 to 7. Therefore the above equation becomes

$$(\mathbf{X})_{u,v} = \frac{1}{4}C(u)C(v)\sum_{i=0}^{7}\sum_{j=0}^{7}(\mathbf{X})_{i,j}\cos\frac{(2i+1)u\pi}{16}\cos\frac{(2j+1)v\pi}{16}, \tag{6.3}$$

where $0 \leq u, v < 7$ and

$$C(u/v) = \begin{cases} \frac{1}{\sqrt{2}} & u/v = 0 \\ 1 & u/v > 0 \end{cases}$$

Direct evaluation of equation 6.4 for an 8x8 DCT (where N = 8) requires $64 * 64 = 4096$ multiply (and the same number of addition) operations. Rearranging the 2D DCT equation 6.3 shows that the 2D DCT can be constructed from two 1D transforms as shown in equation 6.4 [111, 112, 113]. This is also referred as the row/column approach in the literature [114]. The basic idea is that the 2D DCT is calculated by evaluating a 1D DCT for each column of the input matrix (the inner transform), and then evaluating a 1D DCT for each row of the result of the first set of transforms (the outer transform). Each 1D transform takes 64 multiply (and the same number of addition) operations, giving a total of $64 * 8 * 2 = 1024$ multiply (and the same number of addition) operations for an 8 x 8 DCT. This implementation makes the 2D DCT fast compared to a single-pass 2D DCT which takes 4096 multiply (and the same number of addition) operations.

$$(\mathbf{X})_{u,v} = \frac{1}{4}C(v)\sum_{i=0}^{7}\left[C(u)\sum_{j=0}^{7}(\mathbf{X})_{i,j}\cos\frac{(2i+1)u\pi}{16}\right]\cos\frac{(2j+1)v\pi}{16} \tag{6.4}$$

This conversion of a single-pass 2D DCT to two 1D DCTs is not sufficient to reduce the

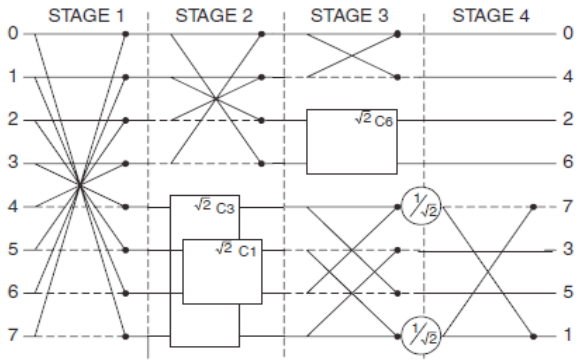


Figure 6.18: Loeffler Algorithm to compute DCT

number of computations. Instead many researchers have proposed a number of algorithms for more efficient computation of this transformation such as Lee [115], Chen [116] , Loeffler [117] and van Eijdhoven [118]. The main objective of these algorithms is to reduce the number of multiplications and additions. We have selected Loeffler's algorithm [117], which is one of the most computationally efficient 1D DCT algorithm as compared to other approaches. It consists of 11 multiplications and 29 additions. This algorithm to calculate a 1D DCT of length 8 is illustrated in figure 6.18. Figures 6.19 and 6.20 explain symbols used in the algorithm

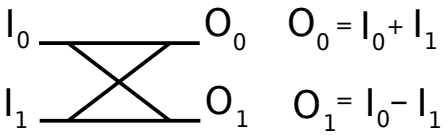


Figure 6.19: The Butterfly Block

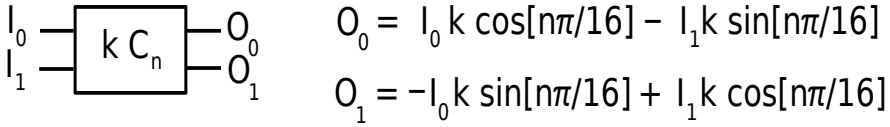


Figure 6.20: The Rotator Block

in figure 6.18. Figure 6.19 shows a butterfly block and the corresponding equations. The rectangular block represents a rotation, which operates on a pair of inputs $[I_0, I_1]$ and produces a pair of outputs $[O_0, O_1]$, as shown in figure 6.20. The constant C_n is equal to $cos\left[\frac{n\pi}{\sqrt{16}}\right]$ or $sin\left[\frac{n\pi}{\sqrt{16}}\right]$. The rotator block can be computed using only 3 multiplications and 3 additions instead of 4 multiplications and 2 additions using the equivalence shown in equation 6.5.

$$\mathbf{O}_0 = aI_0 + bI_1 = (b-a)\, I_1 + a\,(I_0 + I_1)\,, \tag{6.5}$$

$$\mathbf{O}_1 = -bI_0 + aI_1 = -\,(b+a)\, I_1 + a\,(I_0 + I_1)$$

where

$$\mathbf{a} = kcos\left[\frac{n\pi}{16}\right] \text{ and } b = ksin\left[\frac{n\pi}{16}\right]$$

This algorithm consist of 4 stages that have to be computed in serial mode because of the data dependency. However, computations in the individual stages can be done in parallel. In stage 2, the algorithm is split in two parts, one for the even coefficients and one for the odd ones. Figure 6.18 shows that even part of stage 2 is a simple butterfly operation, again separated into even and odd parts in stage 3. The round circle surrounding $\frac{1}{\sqrt{2}}$ means multiplication by $\frac{1}{\sqrt{2}}$.

This algorithm also proposes two approaches for the computation of 2D DCT, using 1D DCT twice and directly using 2D DCT. The former is used due to less computations [119], as described earlier.

Loeffler's algorithm uses fixed/floating point arithmetic for the calculation of DCT. To change it to integer arithmetic, the ISO/IEC standard 23002-1 [120] is used. This standard provides a way to calculate an accurate DCT using integer arithmetic by the use of bit-wise operators. Multiplication and division operators are replaced by shift operators. This is described in section 6.4.3 with the help of the CAPH code to implement this algorithm. For an input image consisting of 8 bits for each pixel, 32 bits are needed to accurately implement DCT using this standard.

6.4.2.2 Quantization

After DCT, the next step to be applied to each 8x8 block is quantization. The goal of this step is to discard data which is not visually significant by taking advantage of the low sensitivity of eye to reconstruction errors related to high frequencies as compared to low frequencies [121].

Quick high frequency changes can often not be seen and can be eliminated whereas slow linear changes in intensity or color are detected by the eye. The basic idea of quantization is to remove as many as the possible nonzero DCT coefficients corresponding to high frequency components. This is accomplished by the division of each block element by its corresponding quantizer step size which is then rounded to nearest integer as shown in equation 6.6.

$$\mathbf{X}^{Q}_{(u,v)} = Round\left(\frac{X_{(u,v)}}{Q_{(u,v)}}\right) \tag{6.6}$$

Varying levels of image compression and quality can be obtained through the selection of a specific quantization matrices. Normally, the standard matrix given in equation 6.7 with quality level of 50 is used. This matrix gives high compression as well as excellent decompressed image quality. Subjective experiments involving the human visual systems have resulted in this standard quantization matrix [122]. In case another level of quality and compression is required, scalar multiples of standard matrix are used. For a quality level greater than 50 (less compression, higher quality image), the standard matrix is multiplied by (100-quality_level)/50. For a quality level less than 50 (more compression, lower image quality), the standard matrix is multiplied by 50/quality_level. The contents of this matrix are totally independent of the input image.

$$\mathbf{Q}_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \tag{6.7}$$

On hardware platforms, especially FPGAs, division utilizes more resources than multiplication. So, the best solution is to accomplish quantization using multiplication rather by division. The quantization equation 6.6 has Q as divisor. This enables the calculation of the table containing values of 1/Q for each possible Q value. Using this table, quantization is done by multiplication instead of division. In the first step, the DCT coefficient is multiplied by a value corresponding to 1/Q in the table. Then the computed value is shifted to the right 16 times to obtain the final result. For example, division by Q value *16* is achieved by first multiplying with the corresponding value in the table i.e. 4096. This value is then shifted on the right 16 times to obtain final result. This replacement of division by multiplication and shift operator comes at a cost. The results obtained are not always as accurate as obtained by division. For example, if the value to be quantized is 5000 and the corresponding value in the quantization matrix is 5, the by using division this will result in 1000 but by using the above method this results in 999.

6.4.2.3 ZigZag Scan

After quantization, most of the coefficients of the block are equal to zero. Each block is then scanned in a zigzag pattern as shown in figure 6.21. The advantage of this scan is to combine a large run of zeros which will compress well in the next step.

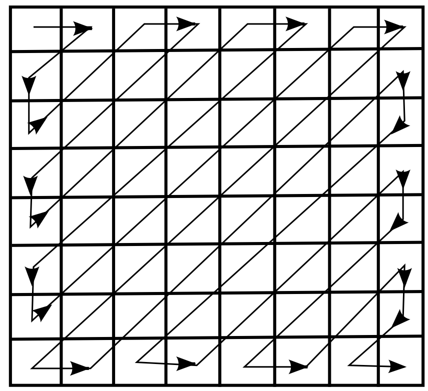


Figure 6.21: ZigZag Scan Pattern

6.4.2.4 Run Length Encoding

After the zigzag scan, the block is encoded (compressed) using two values (LEVEL,RUN). RUN is the distance between two nonzero coefficients i.e. number of zeros in a segment. LEVEL is the nonzero value immediately following a sequence of zeros. The value (0,0) represents end of block(EOB). This is useful in a situation where only zeros are left in the block, so instead of counting all those zeros, this value will mark end of block. This step results in a compressed 8x8 block because after quantization there are large number of zeros in the block and the zigzag ordering groups them together. Consider the input block consists of

26 3 1 3 2 6 2 5 0 2 0 0 0 1 5 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

After applying RLE, the output will be reduce to

26 0 3 0 1 0 3 0 2 0 6 0 2 0 5 1 2 3 1 0 5 5 1 0 0

6.4.3 CAPH implementation

The input to the CAPH design for the JPEG encoder is a sequence of image frames, where each frame consists of a sequence of lists of 8x8 blocks. The CAPH code for the implementation of the JPEG encoder is given in listing 6.15.

Listing 6.15: CAPH implementation of JPEG Encoder

```

----------------------------------------------------------------
------Constant declations
----------------------------------------------------------------
const scale = ( [1024,1138,1730,1609,1024,1609,1730,1138,1138,1264,1922,1788,
1138,1788,1922,1264,1730,1922,2923,2718,1730,2718,2923,1922,1609,1788,2718,
2528,1609,2528,2718,1788,1024,1138,1730,1609,1024,1609,1730,1138,1609,1788,
2718,2528,1609,2528,2718,1788,1730,1922,2923,2718,1730,2718,2923,1922,1138,
1264,1922,1788,1138,1788,1922,1264]:signed<16> array[64]);

const divtab = ([65536,32768,21845,16384,13107,10923,9362,8192,7282,6554,5958,
5461,5041,4681,4369,4096,3855,3641,3449,3277,3121,2979,2849,2731,2621,2521,
2427,2341,2260,2185,2114,2048,1986,1928,1872,1820,1771,1725,1680,1638,1598,
1560,1524,1489,1456,1425,1394,1365,1337,1311,1285,1260,1237,1214,1192,1170,
```

```
15 1150,1130,1111,1092,1074,1057,1040,1024,1008,993,978,964,950,936,923,910,898,
16 886,874,862,851,840,830,819,809,799,790,780,771,762,753,745,736,728,720,712,
17 705,697,690,683,676,669,662,655,649,643,636,630,624,618,612,607,601,596,590,
18 585,580,575,570,565,560,555,551,546,542]:signed<18>array[121]);
19
20 const qtab = ([16,11,10,16,24,40,51,61,12,12,14,19,26,58,60,55,14,13,16,24,40,57,
21 69,56,14,17,22,29,51,87,80,62,18,22,37,56,68,109,103,77,24,35,55,64,81,104,113,
22 92,49,64,78,87,103,121,120,101,72,92,95,98,112,100,103,99]:signed<8>array[64]);
23 ------------------------------------------------------------
24 -----Actors
25 ------------------------------------------------------------
26 actor scale()
27 ...
28 actor stage1 ()
29 ...
30 actor stage2odd ()
31 ...
32 actor stage3odd ()
33 ...
34 actor stage2even ()
35 ...
36 actor stage3even ()
37 ...
38 actor ptos()
39 ...
40 actor transpose()
41 ...
42 actor rightshift()
43 ...
44 actor quant()
45 ...
46 actor zigzag()
47 ...
48 actor rle ()
49 ...
50 -------------------------
51 -- IOs
52 -------------------------
53 stream i:signed<32> dc from "cam:0";
54 stream o:signed<32> dc to "mem:0";
55 -------------------------
56 -- Network declarations
57 -------------------------
58 net dct1d f g h j k l m v =
59 ...
60 net o   = rle zz;
```

The first part of the code consists of constant declarations (lines 5-22).

The CAPH implementation of the JPEG encoder consists of twelve actors, of which, nine are for DCT (i.e. `scale`, `stage1`, `stage2odd`, `stage2even`, `stage34odd`, `stage3even`, `ptos`, `transpose`, `rightshift`), one is for quantization (i.e. `quant`), one is for zigzag scan (i.e. `zigzag`) and one is for Run Length Encoding (i.e. `rle`).

– The `scale` actor performs a left shift on input values by seven places. Functionally

speaking:

$$
\begin{array}{l}
scale :< \\
\quad < p_{11}\ p_{12}\ ...\ p_{18} \\
\quad\ \ p_{21}\ p_{22}\ ...\ p_{28} \\
\quad\quad \vdots \\
\quad\ \ p_{81}\ p_{82}\ ...\ p_{88} > \\
\quad < \quad ... \quad > \\
\quad\quad \vdots \\
>
\end{array}
\longrightarrow
\begin{array}{l}
< \\
\quad < p_{11} \ll 7\ \ p_{12} \ll 7\ \ ...\ p_{18} \ll 7 \\
\quad\ \ p_{21} \ll 7\ \ p_{22} \ll 7\ \ ...\ p_{28} \ll 7 \\
\quad\quad \vdots \\
\quad\ \ p_{81} \ll 7\ \ p_{82} \ll 7\ \ ...\ p_{88} \ll 7\ \ > \\
\quad < \quad\quad ... \quad\quad > \\
\quad\quad \vdots \\
>
\end{array}
$$

The CAPH description of this actor is given in listing 6.16. In the case of a data token at input, it is left shifted seven places and the result is written to the output. If input is a control token, the same token is written to the output.

Listing 6.16: Scale actor

```
1 actor scale()
2   in  (a:signed<32> dc)
3   out (c:signed<32> dc)
4 rules a  ->  c
5 |'<  ->  '<
6 |'>  ->  '>
7 |'p  ->  '(p<<7)
8 ;
```

– The `stage1` actor applies a butterfly block to eight values of a block and gives us eight output values. Functionally speaking :

$$
\begin{array}{l}
stage1 :< \\
\quad < p_{11}\ p_{12}\ ...\ p_{18} \\
\quad\ \ p_{21}\ p_{22}\ ...\ p_{28} \\
\quad\quad \vdots \\
\quad\ \ p_{81}\ p_{82}\ ...\ p_{88} > \\
\quad < \quad ... \quad > \\
\quad\quad \vdots \\
>
\end{array}
\longrightarrow
\begin{array}{l}
< \\
\quad < f(p_{11}\ p_{12}\ ...\ p_{18}) \\
\quad\ \ f(p_{21}\ \ p_{22}\ \ ...\ p_{28}) \\
\quad\quad \vdots \\
\quad\ \ f(p_{81}\ \ p_{82}\ \ ...\ p_{88})\ \ > \\
\quad < \quad\quad ... \quad\quad > \\
\quad\quad \vdots \\
>
\end{array}
$$

where $f(p_{i1}, p_{i2}..., p_{i8}) = p_{i1}+p_{i8}, p_{i1}-p_{i8}, p_{i3}+p_{i6}, p_{i3}-p_{i6}, p_{i2}+p_{i7}, p_{i2}-p_{i7}, p_{i4}+p_{i5}, p_{i4}-p_{i5}$

The CAPH implementation of this actor is given in listing 6.17. To apply the butterfly block, eight input values are required. This is accomplished by saving input values in array `z` in the sixth rule (line 14). After reading eight input values, the butterfly block is applied to the last rule (lines 15-16) and results are written to the output.

Listing 6.17: Stage1 actor

```
actor stage1 ()
  in  (a:signed<32> dc)
  out (c:signed<32> dc,d:signed<32> dc,e:signed<32> dc,f:signed<32> dc,
       g:signed<32> dc,h:signed<32> dc,j:signed<32> dc,k:signed<32> dc)
var s : {S0,S1,S2,S3}=S0
var z : signed<32> array[8] = [0 : 8]
var i : unsigned<8>
rules (s, a, i, z) -> (s, c, d, e, f, g, h, j, k, i, z)
| (S0,'<,_, _) -> (S1,'<,'<,'<,'<,'<,'<,'<,'<, _, _)
| (S1,'>,_, _) -> (S0,'>,'>,'>,'>,'>,'>,'>,'>, _, _)
| (S1,'<,_, z) -> (S2,'<,'<,'<,'<,'<,'<,'<,'<, 0, z[i in 0..7 <-0])
| (S2, _,8, _) -> (S3, _, _, _, _, _, _, _, _, 0, _)
| (S2,'>,_, _) -> (S1,'>,'>,'>,'>,'>,'>,'>,'>, _, _)
| (S2,'v,i, z) -> (S2, _, _, _, _, _, _, _, _,i+1,z[i<-v])
| (S3, _,_, z) -> (S2,'z[0]+z[7],'z[0]-z[7],'z[2]+z[5],'z[2]-z[5],
                    'z[1]+z[6],'z[1]-z[6],'z[3]+z[4],'z[3]-z[4],0 ,_)
;
```

– The `stage2odd` actor reads four input values and calculates four output values by applying Loeffler's algorithm formulas. Functionally speaking :

stage2odd :

$$
\begin{array}{ll}
<< p_{11}\ p_{21}\ ...\ p_{81} > & << f(p_{11},p_{17})\ f(p_{21},p_{27})\ ...\ f(p_{81},p_{87}) > \\
\quad ... <\ ...\ >>, & \quad ... < ... >>, \\
<< p_{13}\ p_{23}\ ...\ p_{83} > & << g(p_{13},p_{15})\ g(p_{23},p_{25})\ ...\ g(p_{83},p_{85}) > \\
\quad ... <\ ...\ >>, \quad \longrightarrow & \quad ... < ... >>, \\
<< p_{15}\ p_{25}\ ...\ p_{85} > & << h(p_{15},p_{13})\ h(p_{25},p_{23})\ ...\ h(p_{85},p_{83}) > \\
\quad ... <\ ...\ >>, & \quad ... < ... >>, \\
<< p_{17}\ p_{27}\ ...\ p_{87} > & << i(p_{17},p_{11})\ i(p_{27},p_{21})\ ...\ i(p_{87},p_{81}) > \\
\quad ... <\ ...\ >> & \quad ... < ... >>
\end{array}
$$

where

$$
\begin{aligned}
f(p_{i1},p_{i7}) &= (((P_{11} \gg 9 - P_{11}) \gg 2)\ -\ (P_{11} \gg 9 - P_{11})) - (P_{17} \gg 1)\ , \\
g(p_{i3},p_{i5}) &= ((P_{13} - (P_{13} \gg 3) - (P_{13} \gg 7)) + ((P_{15} \gg 3) - (P_{15} \gg 7)) + \\
&\qquad ((((P_{15} \gg 3) - (P_{15} \gg 7)) - (P_{15} \gg 11)) \gg 1)), \\
h(p_{i5},p_{i3}) &= ((P_{15} - (P_{15} \gg 3) - (P_{15} \gg 7)) + ((P_{13} \gg 3) - (P_{13} \gg 7)) + \\
&\qquad ((((P_{13} \gg 3) - (P_{13} \gg 7)) - (P_{13} \gg 11)) \gg 1)), \\
i(p_{i7},p_{i1}) &= (((P_{17} \gg 9 - P_{17}) \gg 2)\ -\ (P_{17} \gg 9 - P_{17}))\ -\ (P_{11} \gg 1)
\end{aligned}
$$

The CAPH implementation of this actor is given in listing 6.18. For the data tokens above, the given formulas are applied, otherwise the same control token is written to the output.

Listing 6.18: Stage2 odd actor

```
actor stage2odd ()
  in  (a:signed<32> dc,b:signed<32> dc,c:signed<32> dc,d:signed<32> dc)
```

```
3   out (e:signed<32> dc, f:signed<32> dc, g:signed<32> dc, h:signed<32> dc)
4 rules (a, b, c, d) -> (e, f, g, h)
5 | ('<,'<,'<,'<) -> ('<, '<, '<, '<)
6 | ('>,'>,'>,'>) -> ('>, '>, '>, '>)
7 | ('v1, 'v3,'v5,'v7) ->('(((v1>>9-v1)>>2)-(v1>>9-v1))-(v7>>1),
8                        '((v3-(v3>>3)-(v3>>7))+((v5>>3)-(v5>>7))+
9                         ((((v5>>3)-(v5>>7))-(v5>>11))>>1) ),
10                       '((v5-(v5>>3)-(v5>>7))-(((v3>>3)-(v3>>7))+
11                        ((((v3>>3)-(v3>>7))-(v3>>11))>>1)) ),
12                       '(((v7>>9-v7)>>2)-(v7>>9-v7))+(v1>>1) )
13 ;
```

– The `stage3odd` actor reads four input values and calculates four output values by applying Loeffler's algorithm formulas. Functionally speaking :

stage3odd :

$$
\begin{array}{lcl}
<< p_{11}\ p_{21}\ \dots\ p_{81} > & & << p_{11}+p_{13}+p_{15}+p_{17}\ \dots\ p_{81}+p_{83}+p_{85}+p_{87} > \\
\quad \dots < \dots >>, & & \quad \dots < \dots >>, \\
<< p_{13}\ p_{23}\ \dots\ p_{83} > & & << p_{11}-p_{13}\ \dots\ p_{81}-p_{83} > \\
\quad \dots < \dots >>, & \longrightarrow & \quad \dots < \dots >>, \\
<< p_{15}\ p_{25}\ \dots\ p_{85} > & & << p_{17}-p_{15}\ \dots\ p_{87}-p_{85} > \\
\quad \dots < \dots >>, & & \quad \dots < \dots >>, \\
<< p_{17}\ p_{27}\ \dots\ p_{87} > & & << p_{11}+p_{13}-p_{15}-p_{17}\ \dots\ p_{81}+p_{83}-p_{85}-p_{87} > \\
\quad \dots < \dots >> & & \quad \dots < \dots >>
\end{array}
$$

The implementation is given in listing 6.19. For data tokens above, the given formulas are applied, otherwise the same control token is written to the output.

Listing 6.19: Stage3 odd actor

```
1 actor stage3odd ()
2   in  (a:signed<32> dc, b:signed<32> dc, c:signed<32> dc, d:signed<32> dc)
3   out (e:signed<32> dc, f:signed<32> dc, g:signed<32> dc, h:signed<32> dc)
4 rules (a, b, c, d) -> (e, f, g, h)
5 | ('<,'<,'<,'<) -> (          '<,    '<,   '<,          '<)
6 | ('>,'>,'>,'>) -> (          '>,    '>,   '>,          '>)
7 | ('v1, 'v3,'v5,'v7) -> ('v1+v3+v7+v5, 'v1-v3, 'v7-v5, 'v1+v3-v7-v5)
8 ;
```

– The `stage2even` actor reads four input values and calculates four output values by ap-

plying Loeffler's algorithm formulas. Functionally speaking :

*stage*2*even* :

$$
\begin{array}{ll}
<< p_{12}\ p_{22}\ \ldots\ p_{82} > & << p_{12}+p_{14}+p_{16}+p_{18}\ \ldots\ p_{82}+p_{84}+p_{86}+p_{88} > \\
\quad \ldots < \ \ldots\ >>, & \quad \ldots < \ldots >>, \\
<< p_{14}\ p_{24}\ \ldots\ p_{84} > & << p_{16}-p_{18}\ \ldots\ p_{86}-p_{88} > \\
\quad \ldots < \ \ldots\ >>, \quad \longrightarrow & \quad \ldots < \ldots >>, \\
<< p_{16}\ p_{26}\ \ldots\ p_{86} > & << p_{18}-p_{12}\ \ldots\ p_{88}-p_{82} > \\
\quad \ldots < \ \ldots\ >>, & \quad \ldots < \ldots >>, \\
<< p_{18}\ p_{28}\ \ldots\ p_{88} > & << p_{12}+p_{18}-p_{14}-p_{16}\ \ldots\ p_{82}+p_{88}-p_{84}-p_{86} > \\
\quad \ldots < \ \ldots\ >> & \quad \ldots < \ldots >>
\end{array}
$$

The CAPH implementation of this actor is given in listing 6.20. For data tokens above, the given formulas are applied, otherwise the same control token is written to the output.

Listing 6.20: Stage2 even actor

```
1 actor stage2even ()
2   in (a:signed<32> dc,b:signed<32> dc,c:signed<32> dc,d:signed<32> dc)
3   out(e:signed<32> dc,f:signed<32> dc,g:signed<32> dc,h:signed<32> dc)
4 rules ( a, b, c, d)->( e, f, g, h)
5 | ('<,'<,'<,'<)->(          '<,   '<,       '<,    '<)
6 | ('>,'>,'>,'>)->(          '>,   '>,       '>,    '>)
7 | ('v0, 'v2,'v4,'v6)->('v0+v6+v4+v2,'v4-v2,'v0+v6-v4-v2,'v0-v6)
8 ;
```

– The `stage3even` actor reads four input values and calculates four output values by applying Loeffler's algorithm formulas. Functionally speaking :

*stage*3*even* :

$$
\begin{array}{ll}
<< p_{12}\ p_{22}\ \ldots\ p_{82} > & << p_{12}\ p_{22}\ \ldots\ p_{82} > \\
\quad \ldots < \ \ldots\ >>, & \quad \ldots < \ldots >>, \\
<< p_{14}\ p_{24}\ \ldots\ p_{84} > & << f(p_{14},p_{18})\ f(p_{24},p_{28})\ \ldots\ f(p_{84},p_{88}) > \\
\quad \ldots < \ \ldots\ >>, \quad \longrightarrow & \quad \ldots < \ldots >>, \\
<< p_{16}\ p_{26}\ \ldots\ p_{86} > & << p_{16}\ p_{26}\ \ldots\ p_{86} > \\
\quad \ldots < \ \ldots\ >>, & \quad \ldots < \ldots >>, \\
<< p_{18}\ p_{28}\ \ldots\ p_{88} > & << g(p_{14},p_{18})\ g(p_{24},p_{28})\ \ldots\ g(p_{84},p_{88}) > \\
\quad \ldots < \ \ldots\ >> & \quad \ldots < \ldots >>
\end{array}
$$

$$
\begin{aligned}
&\textit{where}\\
&f(p_{i4}, p_{i8}) = (P_{18} + (P_{18} \gg 5) - (((P_{18} + (P_{18} \gg 5)) \gg 2)))+\\
&\qquad\qquad (((P_{14} + (P_{14} \gg 5)) \gg 2) + (P_{14} \gg 4)),\\
&g(p_{i4}, p_{i8}) = (((P_{18} \ + \ (P_{18} \gg 5)) \gg 2) \ + \ (P_{18} \gg 4)) \ -\\
&\qquad\qquad (P_{14} + (P_{14} \gg 5) - (((P_{14} + (P_{14} \gg 5)) \gg 2)))
\end{aligned}
$$

The CAPH implementation of this actor is given in listing 6.21. It will read input tokens and for data tokens above, the given formulas are applied, otherwise the same control token is written to the output.

Listing 6.21: Stage3 even actor

```
1  actor stage3even ()
2    in (a:signed<32> dc,b:signed<32> dc,c:signed<32> dc,d:signed<32> dc)
3    out(e:signed<32> dc,f:signed<32> dc,g:signed<32> dc,h:signed<32> dc)
4  rules ( a, b, c, d) -> ( e, f, g, h)
5  | ('<,'<,'<,'<) -> (  '<,   '<,   '<,    '<)
6  | ('>,'>,'>,'>) -> (  '>,   '>,   '>,    '>)
7  | ('v0, 'v2,'v4,'v6) -> ( 'v0 ,
8                           '(v6+(v6>>5)-(((v6+(v6>>5))>>2)))+
9                            (((v2+(v2>>5))>>2)+(v2>>4)) ,
10                          'v4 ,
11                          '(((v6+(v6>>5))>>2)+(v6>>4))-
12                           (v2+(v2>>5)-(((v2+(v2>>5))>>2))) )
13 ;
```

– The `ptos` actor acts as a parallel to serial converter. It reads eight input values and sends these value to one output in serial order. Functionally speaking :

ptos :

$$
\begin{array}{lcl}
<< p_{11}\ p_{21}\ ...\ p_{81} >\ ...\ <\ ...\ >>, & & << p_{11}\ p_{12}\ ...\ p_{18}\\
<< p_{12}\ p_{22}\ ...\ p_{82} >\ ...\ <\ ...\ >>, & & \quad p_{21}\ p_{22}\ ...\ p_{28}\\
<< p_{13}\ p_{23}\ ...\ p_{83} >\ ...\ <\ ...\ >>, & & \quad p_{31}\ p_{32}\ ...\ p_{38}\\
<< p_{14}\ p_{24}\ ...\ p_{84} >\ ...\ <\ ...\ >>, & \longrightarrow & \quad p_{41}\ p_{52}\ ...\ p_{58}\\
<< p_{15}\ p_{25}\ ...\ p_{85} >\ ...\ <\ ...\ >>, & & \quad p_{51}\ p_{52}\ ...\ p_{58}\\
<< p_{16}\ p_{26}\ ...\ p_{86} >\ ...\ <\ ...\ >>, & & \quad p_{61}\ p_{62}\ ...\ p_{68}\\
<< p_{17}\ p_{27}\ ...\ p_{87} >\ ...\ <\ ...\ >>, & & \quad p_{71}\ p_{72}\ ...\ p_{78}\\
<< p_{18}\ p_{28}\ ...\ p_{88} >\ ...\ <\ ...\ >>, & & \quad p_{81}\ p_{82}\ ...\ p_{88} >\\
 & & ...\ <\ ...\ >>
\end{array}
$$

This CAPH implementation of this actor is given in listing 6.22. The eight input values will be stored in array `z` in the fifth rule (lines 13-14). In the last rule (line 16) these values are sent to the output `k` one by one by reading the array `z`.

Listing 6.22: Parallel to serial converter actor

```
1  actor ptos()
2    in  (a:signed<32> dc,b:signed<32> dc,c:signed<32> dc,d:signed<32> dc,
3          e:signed<32> dc,f:signed<32> dc,g:signed<32> dc,h:signed<32> dc)
4    out (k:signed<32> dc)
5  var s : {S0,S1,S2,S3}=S0
6  var z : signed<32> array[8] = [0 : 8]
7  var i : unsigned<8>
8  rules (s, a, b, c, d, e, f, g, h, i, z) -> (s, k, i, z)
9  | (S0, '<, '<, '<, '<, '<, '<, '<, '<, _, _) -> (S1,'<, _,        _)
10 | (S1, '>, '>, '>, '>, '>, '>, '>, '>, _, _) -> (S0,'>, _,        _)
11 | (S1, '<, '<, '<, '<, '<, '<, '<, '<, _, z) -> (S2,'<,0,z[i in 0..7 <-0])
12 | (S2, '>, '>, '>, '>, '>, '>, '>, '>, _, _) -> (S1,'>, _,        _)
13 | (S2, 'v0, 'v1, 'v2, 'v3, 'v4, 'v5, 'v6, 'v7, i, z) -> (S3,_,0,
14                          z[0<-v0,1<-v1,2<-v2,3<-v3,4<-v4,5<-v5,6<-v6,7<-v7])
15 | (S3, _, _, _, _, _, _, _, _, 8, _) -> (S2, _, 0,      _)
16 | (S3, _, _, _, _, _, _, _, _, i, z) -> (S3, 'z[i], i+1,      _)
17 ;
```

– The `transpose` actor transposes an 8x8 block. It first reads 64 values and then these values are sent out in an order that will make a transpose of the input block. Functionally speaking :

$$
\begin{array}{l}
transpose :< \\
\quad < p_{11}\ p_{12}\ \cdots\ p_{18} \\
\quad\ \ p_{21}\ p_{22}\ \cdots\ p_{28} \\
\quad\ \ \vdots \\
\quad\ \ p_{81}\ p_{82}\ \cdots\ p_{88} > \\
\quad < \quad \ldots \quad > \\
\quad\ \ \vdots \\
> \\
\end{array}
\longrightarrow
\begin{array}{l}
< \\
\quad < p_{11}\ p_{21}\ \cdots\ p_{81} \\
\quad\ \ p_{12}\ \ p_{22}\ \ \cdots\ p_{82} \\
\quad\ \ \vdots \\
\quad\ \ p_{18}\ \ p_{28}\ \ \cdots\ p_{88}\ \ > \\
\quad < \quad \ldots \quad > \\
\quad\ \ \vdots \\
> \\
\end{array}
$$

The CAPH implementation of this actor is given in listing 6.23. The 8x8 block is saved in array `z` in the fifth rule (line 15). The last two rules (lines 17-18) send the values stored in the array `z` in an order that will transpose the input block. This is accomplished by the index `l` used to read the array. After reading a value, it is incremented to read the value of next line/column.

Listing 6.23: Transpose actor

```
1  actor transpose()
2    in (a:signed<32> dc)
3    out (c:signed<32> dc)
4  var s : {S0,S1,S2,S3}=S0
5  var z : signed<32> array[64] = [0 : 64]
6  var i : unsigned<8>
7  var k : unsigned<8>
8  var l : unsigned<8>
9  var j : unsigned<8>
10 rules (s, a, i, z, k, l, j) -> (s, c, i, z, k, l, j)
11 | (S0,'<, _, _, _, _, _) -> (S1,    '<,    _,         _,    _,    _,    _)
```

```
12 | (S1,'>, _, _, _, _) -> (S0,  '>,  _,  _,  _,  _)
13 | (S1,'<, _, _, _, _) -> (S2,  '<,  0,  _,  _,  _)
14 | (S2,'>, _, _, _, _) -> (S3,  _,  0,  _,  0,  0)
15 | (S2,'v, i, z, _, _) -> (S2,  _,  i+1,  z[i<-v],  _,  _)
16 | (S3, _,64, _, _, _) -> (S1,  '>,  0,  _,  _,  _)
17 | (S3, _, i, z, 7, l, j) -> (S3,  'z[l], i+1,  _,  0, j+1, j+1)
18 | (S3, _, i, z, k, l, _) -> (S3,  'z[l], i+1,  _, k+1, l+8,  _)
19 ;
```

– The `rightshift` actor normalizes the effect of the left shift operation performed at the start (`scale` actor). Functionally speaking :

$$
\begin{array}{l}
rightshift : < \\
\quad < p_{11}\ p_{12}\ \ldots\ p_{18} \\
\quad\ \ p_{21}\ p_{22}\ \ldots\ p_{28} \\
\quad\ \ \vdots \\
\quad\ \ p_{81}\ p_{82}\ \ldots\ p_{88} > \\
\quad < \quad \ldots \quad > \\
\quad\ \ \vdots \\
> \\
\end{array}
\longrightarrow
\begin{array}{l}
< \\
\quad < f(p_{11})\ f(p_{21})\ \ldots\ f(p_{81}) \\
\quad\ \ f(p_{12})\ \ f(p_{22})\ \ \ldots f(p_{82}) \\
\quad\ \ \vdots \\
\quad\ \ f(p_{18})\ \ f(p_{28})\ \ \ldots f(p_{88}) > \\
\quad < \quad \ldots \quad > \\
\quad\ \ \vdots \\
> \\
\end{array}
$$

$$where\ f(P_i) = (((p * scale[i]) + 524287 - (p >> 31)) >> 20)$$

The CAPH implementation of this actor is given in listing 6.24. It first multiplies each element with the corresponding value stored in the array `scale` and then further calculations (addition, subtraction and right shift) are performed to calculate the final value. The array index `i` in last rule (line 11) is used to keep track of the index of elements in the block.

Listing 6.24: Right shift actor

```
1  actor rightshift()
2    in (a:signed<32> dc)
3    out (c:signed<32> dc)
4  var s : {S0,S1,S2}=S0
5  var i : unsigned<8>
6  rules ( s, a , i, scale) -> ( s, c, i)
7  | (S0,  '<,  _,  _) -> (S1,  '<,  _)
8  | (S1,  '>,  _,  _) -> (S0,  '>,  _)
9  | (S1,  '<,  _,  _) -> (S2,  '<,  0)
10 | (S2,  '>,  _,  _) -> (S1,  '>,  _)
11 | (S2,  'p,  i,  scale) -> (S2,'(((p*scale[i])+524287-(p>>31))>>20), i+1)
12 ;
```

– The `quant` actor performs the quantization step. Functionally speaking :

$$
\begin{array}{l}
quant :< \\
\quad < p_{11}\ p_{12}\ \cdots\ p_{18} \\
\quad\ \ p_{21}\ p_{22}\ \cdots\ p_{28} \\
\quad\ \ \vdots \\
\quad\ \ p_{81}\ p_{82}\ \cdots\ p_{88} > \\
\quad < \quad \cdots \quad > \\
\\
\quad\ \ \vdots \\
> \\
\end{array}
\longrightarrow
\begin{array}{l}
< \\
\quad < f(p_{11})\ f(p_{21})\ \cdots\ f(p_{81}) \\
\quad\ \ f(p_{12})\ \ f(p_{22})\ \ \cdots\ f(p_{82}) \\
\quad\ \ \vdots \\
\quad\ \ f(p_{18})\ \ f(p_{28})\ \ \cdots\ f(p_{88}) > \\
\quad < \quad \cdots \quad > \\
\\
\quad\ \ \vdots \\
> \\
\end{array}
$$

$$
where\ f(P_i) = \begin{cases} (((p * divtab[qtab[i] - 1]) >> 16))\ if\ p > 0 \\ (((p * (-1) * divtab[qtab[i] - 1]) >> 16) * (-1))\ otherwise \end{cases}
$$

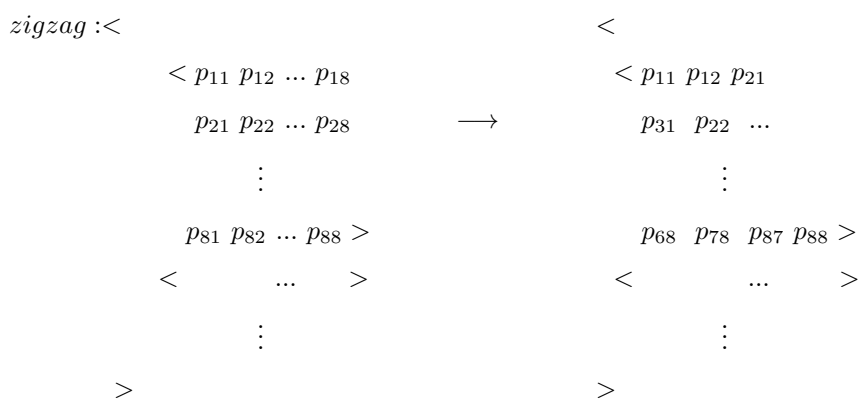
The CAPH implementation of this actor is given in listing 6.25. The last rule (lines 11-14) reads the input value and performs the multiplication and shift operations by reading values from arrays declared in constant declarations to calculate the final result.

Listing 6.25: Quantization actor

```
1  actor quant()
2    in (a:signed<32> dc)
3    out (c:signed<32> dc)
4  var s : {S0,S1,S2}=S0
5  var i : unsigned<8>
6  rules ( s, a , i, divtab, qtab) -> ( s, c, i)
7  | (S0 , '< , _ ,      _ ,     _) -> (S1 , '< ,  _)
8  | (S1 , '> , _ ,      _ ,     _) -> (S0 , '> ,  _)
9  | (S1 , '< , _ ,      _ ,     _) -> (S2 , '< ,  0)
10 | (S2 , '> , _ ,      _ ,     _) -> (S1 , '> ,  _)
11 | (S2 , 'p , i , divtab , qtab) -> (S2 ,
12                                      if(p>0) then '((p * divtab[qtab[i]-1])>>16)
13                                        else '(((p*(-1)*divtab[qtab[i]-1])>>16)*(-1)) ,
14                                      i+1)
15 ;
```

– The `zigzag` actor reads a block and outputs it in zigzag order as shown in figure 6.21.

Functionally speaking :

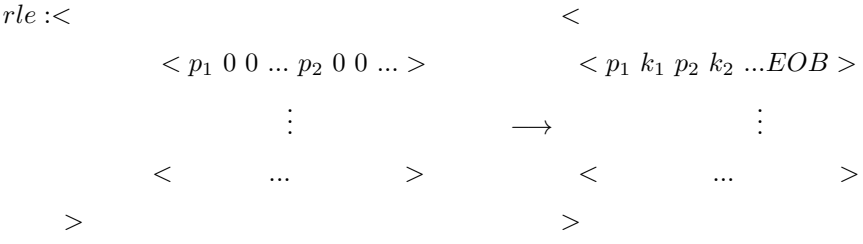


The implementation of this actor is given in listing 6.26. The input block is stored in an array `z` in the fifth rule (line 15). The index at which the element is stored is determined by another array `y`. The array `y` contains the zigzag order. The last rule (line 17) writes to the ouput the block stored in array `z` in zigzag order.

Listing 6.26: Zigzag actor

```
1  actor zigzag()
2    in (a:signed<32> dc)
3    out (c:signed<32> dc)
4  var s : {S0,S1,S2,S3}=S0
5  var z : signed<32> array[64] = [ 0 : 64]
6  var i : unsigned<8>
7  var y : signed<7> array[64] = [0,1,5,6,14,15,27,28,2,4,7,13,16,26,29,42,3,8,
8  12,17,25,30,41,43,9,11,18,24,31,40,44,53,10,19,23,32,39,45,52,54,20,22,33,
9  38,46,51,55,60,21,34,37,47,50,56,59,61,35,36,48,49,57,58,62,63]
10 rules (s, a, i, z, y) -> (s, c, i, z, y)
11 | (S0, '<, _, _, _) -> (S1, '<, _, _, _)
12 | (S1, '>, _, _, _) -> (S0, '>, _, _, _)
13 | (S1, '<, _, _, _) -> (S2, '<, 0, _, _)
14 | (S2, '>, _, _, _) -> (S3, _, 0, _, _)
15 | (S2, 'v, i, z, y) -> (S2, _, i+1, z[y[i]<-v], y)
16 | (S3, _, 64, _, _) -> (S1, '>, 0, _, _)
17 | (S3, _, i, z, _) -> (S3, 'z[i], i+1, _, _)
18 ;
```

– The `rle` actor reads the input block and performs Run Length Encoding (RLE) on it. Functionally speaking :



where k_i is the number of zeros

and EOB (End of block) means the remaining elements in the block are all zero.

The CAPH implementation of this actor is given in listing 6.27. The input block is stored in array `z` in line 19. The index of the End of Block (EOB) is calculated in the 8th rule (lines 21-23). Next, the block elements are written in the form of (LEVEL,RUN) in array `y` in the four rules in lines 29-34. EOB is added in two rules in lines 36-37. Finally, the run length encoded block in array `y` is written to the output in the last rule (lines 39-40).

Listing 6.27: RLE actor

```
1  actor rle ()
2    in (a:signed<32> dc)
3    out (c:signed<32> dc)
4  var s : {S0,S1,S2,S3,S3a,S4,S4a,S4b,S4c,S5,S6,S6a,S7}=S0
5  var z : signed<32> array[64]  =[0 : 64]
6  var y : signed<32> array[128] =[0 : 128]
7  var i : unsigned<8>
8  var k : unsigned<8>
9  var eob : unsigned<8>
10 var zrl : signed<32>
11 var zero: unsigned<1>
12 rules (s, a, i, k, eob, zero, zrl, z, y) -> (s, c, i, k, eob, zero, zrl, z, y)
13 | (S0,'<,_,_,_,_,  _,  _,_,_) -> (S1,'<,_,_,_,_,  _,  _,_,_)
14 | (S1,'>,_,_,_,  _,  _,  _,_,_) -> (S0,'>,_,_,_,_,  _,  _,_,_)
15 | (S1,'<,_,_,_,  _,  _,  _,z,_) -> (S2,'<,0,_,_,_,  _,  _,_,_)
16 | (S2,  _,64,_,_,  _,  _,z,_) -> (S3,  _,63,_,0,if(z[63]=0)then 1 else 0,
17                                    _,  _,  _)
18 | (S2,'>,_,_,_,  _,  _,  _,_,_) -> (S1,'>,  _,_,_,_,  _,  _,_,_)
19 | (S2,  'v,i,_,  _,  _,  _,z,_) -> (S2,  _,i+1,_,_,_,  _,  _,z[i<-v],_)
20 | (S3,  _,0,_,eob,  _,  _,_,_) -> (S4,  _,  0,_,eob,_,  _,_,_)
21 | (S3,  _,i,_,eob,zero,  _,  z,_) -> (if(zero=1)then S3 else S3a,_,
22                                        if(zero=1)then i-1 else 0,  0,  i,
23                                        if(z[i-1]=0)then 1 else 0,  _,  _,  _)
24 | (S3a,_,i,  k,_,  _,  _,  z,  y) -> (S4,_,i+1,k+1,_,  _,  _,  _,y[0<-z[0]])
25 | (S4,_,64,_,_,  _,  _,  _,  _,  _) -> (S2,_,0,_,  _,  _,  _,  _,  _)
26 | (S4,_,i,_,eob,  _,  _,  z,  _) -> (if(i>eob)then S5 else S4a,_,
27                                      if(i>eob)then 0 else i,  _,  _,
28                                      if(z[i]=0)then 1 else 0,  0,  _,  _)
29 | (S4a,_,64,_,  _,  _,  _,_,_,  _) -> (S2,_,_,  _,  _,  _,  _,  _,  _,  _)
30 | (S4a,_,  i,_,_,zero,  zrl,z,_) -> (if(zero=1)then S4a else S4b,  _,i+1,_,_,
31                                       if(z[i+1]=0)then 1 else 0,zrl+1,  _,  _)
32 | (S4b,_,i,k,  _,  _,zrl,  z,  y) -> (S4c,_,_,_,k+1,_,  _,  _,  _,y[k<-zrl-1])
33 | (S4c,_,i,k,  _,zero,  _,  z,  y) -> (S4,_,  _,k+1,_,if(z[i+1]=0)then 1 else 0,
34                                        0,  _,y[k<-z[i-1]])
35 | (S5,_,_,_,_,  eob,  _,  _,_,_) -> (if(eob<63)then S6 else S7,_,_,_,_,_,_,_,_,_)
36 | (S6,_,  _,  k,  _,  _,  _,  _,y) -> (S6a,  _,  _,k+1,_,_,_,_,_,y[k<-0])
37 | (S6a,_,_,_,k,  _,  _,  _,  _,y) -> (S7,  _,  _,k+1,_,_,_,_,_,y[k<-0])
38 | (S7,_,128,_,_,_,_,  _,  _,  _) -> (S2,_,0,_,  _,  _,  _,  _,  _,  _)
39 | (S7,_,i,k,eob,  _,  _,  _,  y) -> (if(i=k-1)then S2 else S7,'y[i],
40                                      if(i=k-1)then 0 else i+1,_,  _,  _,  _,  _,  _)
41 ;
```

I/O streams are declared in listing 6.28.

Listing 6.28: I/O Streams

```
stream i:signed<32> dc from "cam:0";
stream o:signed<32> dc to "mem:0";
```

Dataflow network is described in listing 6.29. The resulting dataflow graph is shown in figure 6.22. Here "dct1d" is defined as *higher-order wiring function* (described in section 3.2.5). It makes it easier to construct dataflow graph which contains two instances of the aforementioned (as 1D DCT is used twice to compute 2D DCT).

Listing 6.29: Network Declarations

```
net dct1d f g h j k l m v =
  let (s1x0,s1x1,s1x2,s1x3,s1x4,s1x5,s1x6,s1x7) = f v in
  let (s2x1,s2x3,s2x5,s2x7) = g(s1x1,s1x3,s1x5,s1x7) in
  let (s2x0,s2x2,s2x4,s2x6) = h(s1x0,s1x2,s1x4,s1x6) in
  let (s3x1,s3x3,s3x5,s3x7) = j(s2x1,s2x3,s2x5,s2x7) in
  let (s3x0,s3x2,s3x4,s3x6) = k(s2x0,s2x2,s2x4,s2x6) in
  let s = m(s3x0,s3x1,s3x2,s3x3,s3x4,s3x5,s3x6,s3x7) in
  l(s);

net sc  = scale i;
net row =
    dct1d stage1 stage2odd stage2even stage3odd stage3even transpose ptos sc;
net col =
    dct1d stage1 stage2odd stage2even stage3odd stage3even transpose ptos row;
net rs  = rightshift col;
net qt  = quant rs;
net zz  = zigzag qt;
net o   = rle zz;
```

6.4.4 Experimental Results

This section will compare FPGA implementations of JPEG encoder parts using different development methodologies/tools. These methodologies include handwritten VHDL code and automatically generated code from two dataflow compilers, CAPH [79] and CAL [123]. The JPEG encoder is selected as application for comparison because of its complex implementation and intensive computations.

The selected parts of the encoder are first implemented in Matlab. The results obtained by Matlab are used as a reference to validate VHDL results obtained by the above three methodologies. The Matlab and handwritten VHDL implementations are given in Appendix A and B respectively. The best way to compare encoder results is to take a 8x8 block as input and compare the results generated by Matlab with these methodologies. The CAL code for these encoder parts is obtained from [124], written by the CAL development community. This code is compiled using the Eclipse IDE to generate VHDL code.

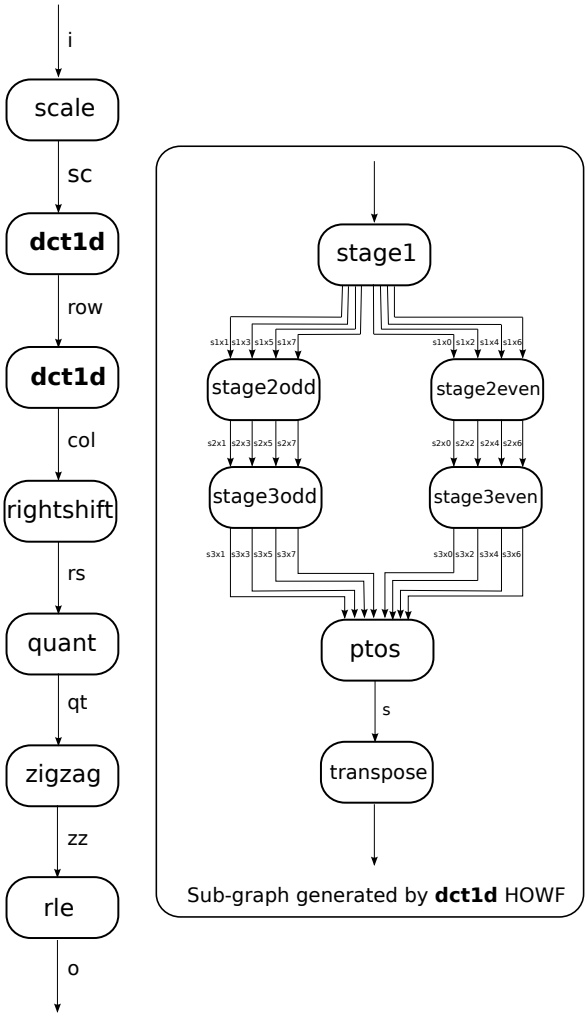


Figure 6.22: Dataflow graph of JPEG encoder application

For our experiments, the following 8x8 block is selected as input is :

154	123	123	123	123	123	123	136
192	180	136	154	154	154	136	110
254	198	154	154	180	154	123	123
239	180	136	180	180	166	123	123
180	154	136	167	166	149	136	136
128	136	123	136	154	180	198	154
123	105	110	149	136	136	180	166
110	136	123	123	123	136	154	136

6.4.4.1 Final Results

Discrete Cosine Transformation(DCT)

First, Loeffler's algorithm is implemented in Matlab using the ISO/IEC 23002-1 standard. Before moving to other implementations, the results obtained are compared with the built in Matlab function *dct2* to calculate 2D DCT. The result obtained by the built in Matlab function *dct2* is:

1186	41	20	72	30	12	-20	-11
30	108	10	32	28	-16	18	-2
-94	-60	12	-43	-31	6	-3	7
-39	-83	-5	-22	-14	15	-1	4
-31	18	-6	-12	14	-6	11	-6
-1	-12	13	0	28	13	8	3
5	-2	12	7	-19	-13	8	12
-10	11	8	-16	21	0	6	11

The result obtained by Matlab implementation of Loeffler's Algorithm using ISO/IEC 23002-1 standard is:

1187	41	21	72	31	13	-19	-11
30	107	11	33	28	-14	19	-1
-94	-59	13	-43	-31	6	-3	7
-38	-81	-5	-22	-13	15	-1	4
-31	18	-5	-12	15	-5	12	-5
0	-10	13	1	28	13	9	3
5	-2	13	7	-18	-12	8	13
-10	11	8	-15	22	1	6	11

As it can be observed, there is a minor difference of results from the two approaches. The variance in result by the use of ISO/IEC 23002-2 is negligible. Afterwards, the same algorithm is implemented using CAPH, CAL and handwritten VHDL.

The result obtained by CAPH is:

1182	35	15	67	26	10	-22	-12
25	99	3	26	22	-19	15	-3
-99	-67	6	-49	-36	2	-6	5
-43	-88	-11	-27	-18	11	-4	2
-35	12	-11	-17	10	-9	9	-7
-4	-15	9	-3	25	10	7	2
3	-5	9	4	-21	-14	7	11
-11	9	6	-17	20	-1	5	10

The result obtained by CAL is:

1186	41	20	72	30	13	-20	-11
30	106	10	32	27	-14	18	-1
-94	-60	12	-43	-31	6	-3	7
-38	-81	-5	-22	-13	15	-1	3
-31	18	-6	-12	14	-6	11	-6
-1	-11	13	1	28	12	8	3
5	-2	12	7	-19	-13	8	12
-10	11	8	-16	21	0	6	10

The result obtained by handwritten VHDL is:

1182	35	15	67	26	10	-22	-12
25	99	3	26	22	-19	15	-3
-99	-67	6	-49	-36	2	-6	5
-43	-88	-11	-27	-18	11	-4	2
-35	12	-11	-17	10	-9	9	-7
-4	-15	9	-3	25	10	7	2
3	-5	9	4	-21	-14	7	11
-11	9	6	-17	20	-1	5	10

By observing the above results, it is clear that results obtained by CAL are more accurate compared to CAPH or handwritten VHDL. The reason is that CAL generated VHDL code involves a lot of data conversions to integer, signed/unsigned, where as in the later two cases, operations are performed on std_logic_vector, without any conversion to integer. Even then the maximum difference is 5 or 6. This is not big difference, as in the next step (i.e.quantization), it will reduce to just 1.

Quantization

The output of DCT is used as input to the quantization. The Matlab implementation of quantization gives the following result:

74	4	2	5	1	0	0	0
2	9	1	2	1	0	0	0
-7	-5	1	-2	-1	0	0	0
-3	-5	0	-1	0	0	0	0
-2	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

The result obtained by CAPH is:

73	3	1	4	1	1	0	0
2	8	0	1	0	0	0	0
-7	-5	0	0	0	0	0	0
-3	-5	-2	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

The result obtained by CAL is:

74	4	2	5	0	1	0	0
2	8	0	1	-1	0	0	0
-6	-4	1	0	0	0	0	0
-2	1	0	0	0	0	0	0
-4	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

The result obtained by handwritten VHDL is:

73	3	1	4	1	1	0	0
2	8	0	1	0	0	0	0
-7	-5	0	0	0	0	0	0
-3	-5	-2	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

By observing the above four results, it clear that they are almost identical. The maximum difference is of 1. The reason of this small difference is the replacement of the division operator by multiplication and shift operators. This has been explained in detail in section 6.4.2.2.

Zigzag Scan

The output of quantization is input to the zigzag scan. The result obtained by the Matlab implementation is :

74	4	2	-7	9	2	5	1
-5	-3	-2	-5	1	2	1	0
1	-2	0	1	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

The result obtained by CAPH is:

73	3	2	-7	8	1	4	0
-5	-3	-1	-5	0	1	1	0
0	-2	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

The result obtained by CAL is:

74	4	2	-6	8	2	5	0
-4	-2	-4	1	1	1	0	1
-1	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

The result obtained by handwritten VHDL is:

73	3	2	-7	8	1	4	0
-5	-3	-1	-5	0	1	1	1
0	0	-2	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

This step only rearranges input values in a specific order. But here results of four implementations are not exactly same because of different input blocks. The actual comparison will be in resource utilization which will be discussed in next section.

Run Length Encoding

The output of the zigzag scan is input to RLE. The result obtained by the Matlab implementation is:

74 0 4 0 2 0 -7 0 9 0 2 0 5 0 1 0 -5 0 -3 0 -2 0 -5 0 1 0 2 0 1 0 1 1 -2 0 1 1 -1 4 -1 0 0 0

The CAPH implemention result is :

73 0 3 0 2 0 -7 0 8 0 1 0 4 0 -5 1 -3 0 -1 0 -5 0 1 1 1 0 -2 2 0 0

The CAL implementation result is :

74 0 4 0 2 0 -6 0 8 0 2 0 5 0 -4 1 -2 0 -4 0 1 0 1 0 1 0 1 1 -1 0 1 2 0 0

The handwritten VHDL implementation result is :

73 0 3 0 2 0 -7 0 8 0 1 0 4 0 -5 1 -3 0 -1 0 -5 0 1 1 1 0 1 0 -2 2 0 0

For the RLE, no arithmetic operations are performed. The result will be identical for the same input block.

6.4.4.2 Performance Results

Altera

Tables 6.4, 6.5 and 6.6 provide a comparison of resources used by CAPH, CAL and handwritten VHDL code for the implementation of the encoder parts for the Altera FPGA.

Table 6.4: DCT (Altera)

	Total	CAPH	CAL	Handwritten VHDL
Max Frequency		53 MHz	43 MHz	57 MHz
Logic Elements	57120	8,637 (15%)	8,607 (15%)	5,992 (10%)
Memory Bits	5,215,104	8,536 (<1%)	12,264 (<1%)	4,096 (<1%)
DSP Blocks	144	8 (6%)	8 (6%)	8 (6%)

Table 6.4 shows experimental results for the implementation of DCT. When comparing CAPH and CAL, CAPH gives a maximum frequency of 52 MHz and CAL 43 MHz. CAPH and CAL use almost the same number of Logic Elements (LEs), 8,637 and 8,607 respectively (15% of total). Memory bits consumed by CAPH are 8,536 and by CAL are 12,264. Here in both cases all memory bits are consumed by FIFOs between actors. The reason CAPH consumed less memory bits than CAL is because of the calculation of the maximum depth needed for each FIFO before generating VHDL code. This is done by calculating the run time occupancy of each FIFO (described in section 4.3.1). The number of DSP blocks used is the same for both, 8 (6% of total). In fact, the DSP block consumption directly depends on the number of multiplication operations used, since it is equal for both codes, so the DSP consumption is same.

When comparing handwritten VHDL and CAPH for DCT, the former generates a maximum frequency of 57 MHz as compared to CAPH's 52 MHz. Handwritten VHDL consumes 6000 LEs (10% of total) as compared to 8,637 by CAPH. Memory bits consumed by handwritten VHDL are 4096, far less than 8,536 consumed by CAPH. The reason CAPH consumes more memory elements is because of the basic principle of dataflow methodology where different actors are connected by FIFOs. In the handwritten VHDL, memory bits are consumed by arrays which are used to store elements. The number of DSP blocks used is the same for both, 8(6% of total).

Table 6.5: Quantization + Zigzag Ordering (Altera)

	Total	CAPH	CAL	Handwritten VHDL
Max Frequency		49 MHz	43 MHz	50 MHz
Logic Elements	57120	600 (1%)	572 (1%)	456 (1%)
Memory Bits	5,215,104	2,048 (<1%)	2,048 (<1%)	2,048 (<1%)
DSP Blocks	144	16 (11%)	16 (11%)	16 (11%)

Quantization and zigzag scan are not as extensive in terms of resources as DCT, so both are combined for performance comparison. Both consist of one actor each. As shown in Table 6.5, CAPH has a maximum frequency of 49 MHz and CAL has 43 MHz. CAPH consume 600 LEs and CAL 572 : both are 1% of total. Since the actors are not complex, LE consumption is almost the same. Only one FIFO needed is before `zigzag` actor and depth of this FIFO is equal to the size of block. So, memory bit consumption is same for both, which is 2048. Here again, because of the same reason discussed earlier, DSP block consumption is 16 for both.

For quantization and zigzag scan, handwritten VHDL gives a maximum frequency of 50

MHz, slightly better than 49 MHz of CAPH. Handwritten VHDL consumes 456 LEs as compared to 600 by CAPH. Actors are simple, so the difference of LE is small. Memory bit consumption is 2,048 for both because both need to store one block for zigzag ordering. Both use the same number of DSP blocks, 16 (11% of total). As this part consumes less resources, so the difference is also small.

Table 6.6: Run Length Encoding (Altera)

	Total	**CAPH**	**CAL**	**Handwritten VHDL**
Max Frequency		75 MHz	68 MHz	88 MHz
Logic Elements	57120	700 (1%)	1,022 (2%)	560 (1%)
Memory Bits	5,215,104	6,144 (<1%)	6,144 (<1%)	6,144 (<1%)
DSP Blocks	144	0 (0%)	0 (0%)	0 (0%)

In the case of RLE, CAPH and CAL have a maximum frequency of 75 MHz and 68 MHz respectively as shown in Table 6.6. LEs consumed by CAPH are 560 (1% of total), whereas CAL consumes 1,022 (2% of total). Both consume 6,144 memory bits : here memory bits are not consumed by a FIFO but by large arrays which are used in the actor to store blocks. For both codes, no DSP block is used in this part as it does not involve any multiplication operation.

For RLE, handwritten VHDL has a maximum frequency of 88 MHz, compared to 75MHz for CAPH. LEs consumed by handwritten VHDL and CAPH are almost the same, at 560 and 700 respectively (1% of total). Both consume 6,144 memory bits, which are used inside the actor to store blocks. Whereas, since no multiplication operation is performed in this part, no DSP block is used by either.

It can be observed from the discussion of the first three tables that the main difference in resources is in DCT implementation and the remaining two parts are almost the same. This is the reason DCT is considered the most important part of a decoder and performance comparisons are drawn on the basis of DCT.

Based upon the above discussion, it can be concluded that on the Altera FPGA, CAPH generated code is slight behind in performance and resource utilization when compared with handwritten VHDL but it is better than CAL in the case of the above implemented parts of encoder.

Xilinx

To compare resource utilization on Xilinx, the HDL code is synthesized using the ISE design software version 10.1. The device selected is a Virtex 2P XC2VP70. Tables 6.7, 6.8 and 6.9 describe resources consumed by each methodology for the implementation of encoder parts.

Table 6.7: DCT (Xilinx)

	Total	**CAPH**	**CAL**	**Handwritten VHDL**
Max Frequency		58 MHz	57 MHz	72 MHz
Slices	33,088	5,582 (16%)	3,955 (11%)	2,745 (8%)
Slice Flip Flops	66,176	5,655 (9%)	3,597 (5%)	2,406 (3%)
4 input LUTs	66,176	9,346 (14%)	5,046 (7%)	4,456 (6%)
Block RAMs	328	5 (1%)	9 (2%)	4 (1%)

For DCT as shown in Table 6.7, CAPH has a maximum frequency of 58 MHz whereas CAL has 57 MHz. The number of slices consumed by CAL is 3,955 (11% of total) and by CAPH 5,582

(16% of total). The number of slice flip flops is 5,655 (9% of total) for CAPH and 3,597(5% of total) for CAL. The consumption of 4 input LUTs is 9,346 (14% of total) for CAPH and 5,046 (7% of total) for CAL. The RAM blocks used by CAPH are 5 (1% of total) and by CAL are 9 (2% of total). It can be deduced that CAL utilizes less resources than CAPH on Xlinix but CAPH has better maximum frequency.

When compared with handwritten VHDL, it has a maximum frequency of 72 MHz, compared with CAPH's 58 MHz. The number of slices consumed by handwritten code is 2,745 (8% of total) and 5,582 (16% of total) by CAPH. In terms of the number of slice flip flops, handwritten VHDL uses 2,406 (3% of total) and CAPH uses 5,655 (9% of total). The total number of 4 input LUTs utilized by handwritten VHDL is 4,456 (6% of total) and for CAPH 9,346 (14% of total). Block RAMs used by Handwritten VHDL is 4 (1% of total) and 5 (1% of total) by CAPH. The consumption of resources is almost the same for both but handwritten VHDL is better in terms of maximum frequency. For quantization and zigzag scan, the results are

Table 6.8: Quantization + Zigzag Ordering (Xilinx)

	Total	**CAPH**	**CAL**	**Handwritten VHDL**
Max Frequency		64 MHz	66 MHz	77 MHz
Slices	33,088	253 (<1%)	286 (<1%)	221 (<1%)
Slice Flip Flops	66,176	227 (<1%)	196 (<1%)	125 (<1%)
4 input LUTs	66,176	500 (<1%)	462 (<1%)	423 (<1%)
Block RAMs	328	1 (<1%)	1 (<1%)	1 (<1%)

shown in Table 6.8. Since this part is not complex, so resource consumption is almost the same for CAPH and CAL. CAL has a maximum frequency of 64 MHz which is almost the same as CAPH's 66 MHz. Both consume less than 1% of the slices, with CAPH using 300 and CAL using 286. The number of slice flip flops used by CAPH is 227 (<1% of total) and by CAL is 196 (<1% of total). The total number of 4 input LUTs consumption is also less than 1% of the total, with CAPH using 500 and CAL using 462. Both use one RAM block.

When comparing CAPH and handwritten VHDL, the later has a better maximum frequency of 77 MHz as compared to the former's 64 MHz. For slices, slice flip flops and 4 input LUTs, resource consumption is almost the same, with both utilizing <1% of each. Both use one RAM block. So, apart from the maximum frequency, the rest of the parameters are the same for CAPH and handwritten VHDL.

Table 6.9: Run Length Encoding (Xilinx)

	Total	**CAPH**	**CAL**	**Handwritten VHDL**
Max Frequency		129 MHz	130 MHz	135 MHz
Slices	33,088	480 (1%)	542 (1%)	380 (1%)
Slice Flip Flops	66,176	500 (<1%)	570 (<1%)	490 (<1%)
4 input LUTs	66,176	937 (1%)	849 (1%)	475 (1%)
Block RAMs	328	2 (<1%)	2 (<1%)	2 (<1%)

In the case of Run Length Encoding(RLE) as shown in Table 6.9, CAPH has a maximum frequency of 129 MHz whereas CAL has 130 MHz. The number of slices consumpted is 480 by CAPH and 542 by CAL (both are 1% of total). CAPH uses 500 slice flip flops and CAL 570, which are <1% of total. The number of 4 input LUTs used by CAPH is 937 and 849 by CAL. Both use two RAM blocks. In this part, the maximum frequency as well as the resource

consumption is almost the same for both languages.

When comparing with handwritten VHDL, it has a better maximum frequency of 135 MHz as compared to 129 MHz by CAPH. In this part also, the consumed number of slices, slice flip flops and 4 input LUTs is almost the same for both language (around 1% of total). The number of block RAMs used by both is 2. For RLE, the only major difference between CAPH and handwritten VHDL is the maximum frequency and resource consumption is almost same.

On Xilinx hardware, CAPH generated VHDL code is behind CAL generated code. The big difference is in the DCT part; in the remaining two parts, the results are almost the same.

Table 6.10 and 6.11 summarize the resource utilization for all encoder parts combined for the three methodologies on Altera and Xilinx FPGAs respectively.

Table 6.10: All parts (Altera)

	Total	**CAPH**	**CAL**	**Handwritten VHDL**
Max Frequency		40 MHz	41 MHz	47 MHz
Logic Elements	57120	9,686 (17%)	10,091 (18%)	6,785 (12%)
Memory Bits	5,215,104	16,728 (<1%)	19,456 (<1%)	12,288 (<1%)
DSP Blocks	144	24 (17%)	24 (17%)	24 (17%)

Table 6.11: All parts (Xilinx)

	Total	**CAPH**	**CAL**	**Handwritten VHDL**
Max Frequency		59 MHz	58 MHz	74 MHz
Slices	33,088	6,680 (20%)	5,232 (15%)	4,515 (13%)
Slice Flip Flops	66,176	6,590 (10%)	4,630 (6%)	3,305 (4%)
4 input LUTs	66,176	11,920 (18%)	7,207 (10%)	6,931 (10%)
Block RAMs	328	8 (2%)	12 (3%)	7 (2%)

On the Altera hardware, CAPH has a maximum frequency of 40 MHz, compared to 41 MHz for CAL. CAPH consumes 9,686 LEs (17% of total), slightly less than 10,091 LEs (18% of total) by CAL. CAPH uses 16,728 meory bits and CAL uses 19,456, both are less than 1% of total. Both use the same number of DSP blocks (i.e. 24). In case of handwritten VHDL, it has a better maximum frequency of 47 MHz, compared to 40 MHz for CAPH. It consumes 6,785 LEs (12% of total) as compared to 9,686 (17% of total) by CAPH. The memory bit consumption is 12,288 , the numbers are less than consumed by CAPH but the percertange of total is the same for both. The number of DSP blocks consumed is same for both methodologies.

On the Xilinx hardware, CAPH has a maximum frequency of 59 MHz, compared to 58 MHz for CAL. CAPH consumes 6,680 slices (20% of total), where as CAL consumes 5,232 (15% of total). Slice flip flops consumtion for CAPH is 6,590 (10% of total) and 4,630 (6% of total) for CAL. The number of 4 input LUTs used by CAPH is 11,920 (18% of total) and 7,207 (10% of total) by CAL. The number of block RAMs used by CAPH is 8 (2% of total) and 12 (3% of total). When comparing with handwritten VHDL, it has a maximum frequency of 74 MHz, compared to 59 MHz by CAPH. Handwritten VHDL uses 4,515 slices (13% of total), whereas CAPH uses 5,232 (15% of total). The slice flip flops consumption is 3,305 (4% of total) for handwritten VHDL and 6,590 (10% of total) for CAPH. Handwritten VHDL consumes 6,931 (10% of total) 4 input LUTs and CAPH 11,920 (18% of total). The number of block RAMs used is 7 for handwritten VHDL and 8 for CAPH, both are 2% of total.

Overall, the handwritten VHDL outperforms the code generated by both the methodologies, both for the maximum frequency as well as resources consumed. But both of these methodologies (i.e. CAPH and CAL) offer a significant gain in programming effort. When considering this effort, the overhead in performance can be neglected. On the other hand, it is quite interesting to compare CAPH and CAL on two different FPGAs. On the Altera FPGA, code generated by CAPH consumes less LEs and memory bits compared to CAL code. On the other hand, for Xilinx FPGA, CAL code consumes less slices, slice flip flops and 4 input LUTs compared to CAPH code. Whereas CAPH consumes less block RAMs due to FIFO size estimation (detailed in section 4.3). The same is the reason for less memory bits consumption on the Altera FPGA. In terms of maximum frequency, on both the platforms, there is no significant difference.

The organization of on-chip memory blocks is different on both FPGAs. Stratix device uses the "TriMatrix" memory structure which consists of three sizes of RAM blocks (512-bit, 4K-bit, 512K-bit). On the other hand, the VirtexII device only provides RAM blocks with a fixed size of 18 Kbit. So the memory utilization is efficient on Stratix compared to VirtexII.

Chapter 7

Conclusion

Over the past decade, FPGAs use has been growing rapidly as a result of advancement in silicon technology. A potential risk to the hindrance of this growth is the availability of sophisticated design tools for implementing applications from high-level descriptions. This thesis evaluates the effectiveness of the *dataflow* programming model for implementing *stream processing* applications on FPGAs.

CAPH, a domain specific language (DSL) based on the dataflow programing model, is selected for this evaluation. Applications are described in CAPH as networks of dataflow actors operating on *structured streams* of tokens. This makes the language very well suited to the description of stream processing applications. The language has a solid formal basis (rooted in functional programming principles) and this eases the development of target-specific back ends.

Our work essentially concerned the VHDL backend (and, marginally the SystemC backend since some by-products of the latter are actually used by the former). A number of applications ranging from very simple to complex have been written in order to explore both the expressivity of the language and the efficiency of the generated VHDL code.

Our results show that, at least for the set of demonstrated applications, CAPH offers a significant improvement in expressivity compared to hand-crafted VHDL while producing code whose performances are on a par with the latter solution.

This said, CAPH is still a "young" language and our work must be viewed as preliminary. As a result, many questions remain open, offering opportunities for further work.

First, the set of applications for assessing the language and associated tools has to be enriched, in size and spectrum, to confirm our conclusions. More comparisons with existing applications (both in VHDL and with other high-level languages) are definitely needed. Ultimately, the question of whether the dataflow model is actually suited to the formulation of all stream-processing applications remains open. An intermediate question concerns the amount of reformulation that a programmer is ready to do in order to adapt a initial formulation, written in a imperative language for example, to this model.

Second, the language and the tools themselves could be improved in several ways. For example, the introduction of *sized types* could make the actor descriptions more "generic" (in the sense that they could operate on tokens having a variable size in bits), this allowing the construction of a fully reusable library of actors. The definition of *higher-order* actors (*i.e.* actors taking functions as parameters) could push this genericity even further. On the other side, the VHDL back end could be optimized to produce even better code. For instance, a limitation, deriving from the current elaboration process, is that the expressions on the right-hand side of the rules are evaluated in one clock cycle. This is not a problem for "simple" actors such as the ones used in the application described here but we anticipate that for actors involving more complex computations, this approach could result in unacceptable critical paths. In this case, the programmer would have to "break" complex actors into small enough actors to reach a given clock frequency. Some work is needed for formalizing and implementing the transformation rules that could assist the programmer in carrying out this transformation.

Third, although CAPH has been used mainly to implement real-time image processing applications, the dataflow, stream-processing model it supports could be applied to many other domains, such as nD-signal processing, number crunching applications or neural network simulations.

Finally, the code will be generated for more hardware elements. For example, for the DSP

co-processor in case of SeeMOS platform. At the start, the code generated for each hardware part will be hard coded in code. Later on, it will be done automatically by calculating the computing time for each part of code on different hardware parts. This approach can be further extended to another research platform which contains more hardware parts or by the addition of hardware elements on SeeMOS platform.

Appendix A

Matlab Code for JPEG Encoder

```
function res = encoder()
A = [154 123 123 123 123 123 123 136
     192 180 136 154 154 154 136 110
     254 198 154 154 180 154 123 123
     239 180 136 180 180 166 123 123
     180 154 136 167 166 149 136 136
     128 136 123 136 154 180 198 154
     123 105 110 149 136 136 180 166
     110 136 123 123 123 136 154 136];

scale = [1024 1138 1730 1609 1024 1609 1730 1138
         1138 1264 1922 1788 1138 1788 1922 1264
         1730 1922 2923 2718 1730 2718 2923 1922
         1609 1788 2718 2528 1609 2528 2718 1788
         1024 1138 1730 1609 1024 1609 1730 1138
         1609 1788 2718 2528 1609 2528 2718 1788
         1730 1922 2923 2718 1730 2718 2923 1922
         1138 1264 1922 1788 1138 1788 1922 1264];

 QTab = [16 11 10 16 24  40  51  61
         12 12 14 19 26  58  60  55
         14 13 16 24 40  57  69  56
         14 17 22 29 51  87  80  62
         18 22 37 56 68  109 103 77
         24 35 55 64 81  104 113 92
         49 64 78 87 103 121 120 101
         72 92 95 98 112 100 103 99];

DivTab = [65536 32768 21845 16384 13107 10923 9362 8192
          7282  6554  5958  5461  5041  4681  4369 4096
          3855  3641  3449  3277  3121  2979  2849 2731
          2621  2521  2427  2341  2260  2185  2114 2048
          1986  1928  1872  1820  1771  1725  1680 1638
          1598  1560  1524  1489  1456  1425  1394 1365
          1337  1311  1285  1260  1237  1214  1192 1170
          1150  1130  1111  1092  1074  1057  1040 1024
          1008  993   978   964   950   936   923  910
          898   886   874   862   851   840   830  819
          809   799   790   780   771   762   753  745
          736   728   720   712   705   697   690  683
```

```
          676   669   662   655   649   643   636  630
          624   618   612   607   601   596   590  585
          580   575   570   565   560   555   551  546 542];

ZZTab = [0   1   8   16  9   2   3  10
         17  24  32  25  18  11  4   5
         12  19  26  33  40  48  41  34
         27  20  13  6   7   14  21  28
         35  42  49  56  57  50  43  36
         29  22  15  23  30  37  44  51
         58  59  52  45  38  31  39  46
         53  60  61  54  47  55  62  63];

% shift left 7
A3 = A*128;

%first 1D DCT
for i=1:8
    OutT(i,:)=f_dct(A3(i,:));
end

%Transpose
B = transpose(OutT);

%Second 1D DCT
for i=1:8
    OutT(i,:)=f_dct(B(i,:));
end

%Transpose
OutT = transpose(OutT);

%shift right + scale
dct = round((OutT .* scale + 524287 - (OutT/2147483648)) / 1048576);

%Quantization
for i =1:8
    for j=1:8
        qtz(i,j) = (dct(i,j) * DivTab(1,QTab(i,j)))/65536;
    end
end

%Transpose+ reshape to make Quantization matrix one dimentioanal array
qtz = reshape(transpose(qtz),1,64);

%ZigZag Ordering
for i =1:8
    for j=1:8
        zz(i,j) = qtz(1,ZZTab(i,j)+1);
    end
end

%Reshape zigzag to 1D array and round to make comparison equal to zero
zz = round(reshape(transpose(zz),1,64));

%Run Length Encoding
EOB_index = 64;
```

```
while (zz(EOB_index)==0)
    EOB_index = EOB_index-1;
end

k=1;
i=1;
while( i <=EOB_index)
    ZRL=0;
    zero=(zz(i)==0);
    while(zero)
        ZRL=ZRL+1;
        i=i+1;
        zero=(zz(i)==0);
    end
    rle(1,k)=zz(i);
    k=k+1;
    rle(1,k)=ZRL;
    i=i+1;
    k=k+1;
end
if(EOB_index<64)
    rle(1,k)=0;
    rle(1,k+1)=0;
end

res = rle;
end

function [OutT] = f_dct(InT)
                x0 = InT(1) + InT(8);
                x1 = InT(1) - InT(8);
                x4 = InT(2) + InT(7);
                x5 = InT(2) - InT(7);
                x2 = InT(3) + InT(6);
                x3 = InT(3) - InT(6);
                x6 = InT(4) + InT(5);
                x7 = InT(4) - InT(5);
                xa = pmul_1_2(x3);
                x3 = pmul_1_1(x3);
                xb = pmul_1_2(x5);
                x5 = pmul_1_1(x5);
                x3 = x3 + xb;
                x5 = x5 - xa;
                xa = pmul_2_2(x1);
                x1 = pmul_2_1(x1);
                xb = pmul_2_2(x7);
                x7 = pmul_2_1(x7);
                x1 = x1 - xb;
                x7 = x7 + xa;
                xa = x1 + x3;
                x3 = x1 - x3;
                xb = x7 + x5;
                x5 = x7 - x5;
                x1 = xa + xb;
                x7 = xa - xb;
                xa = x0 + x6;
                x6 = x0 - x6;
```

```
                xb = x4 + x2;
                x2 = x4 - x2;
                x0 = xa + xb;
                x4 = xa - xb;
                xa = pmul_3_2(x2);
                x2 = pmul_3_1(x2);
                xb = pmul_3_2(x6);
                x6 = pmul_3_1(x6);
                x2 = xb + x2;
                x6 = x6 - xa;
                OutT(1) = x0;
                OutT(2) = x1;
                OutT(3) = x2;
                OutT(4) = x3;
                OutT(5) = x4;
                OutT(6) = x5;
                OutT(7) = x6;
                OutT(8) = x7;
end

function x1 = pmul_1_1(x)
        x1 = x - (x / 8) - (x / 128);
end

function x2 = pmul_1_2(x)
        tmp1 = (x / 8) - (x / 128);
        tmp2 = tmp1 - (x / 2048);
        x2 = tmp1 + (tmp2/2);
end

function x3 = pmul_2_1(x)
        tmp = (x / 512) - x ;
        x3 = (tmp /4) - tmp;
end

function x4 = pmul_2_2( x)
        x4 = x /2;
end

function x5 = pmul_3_1(x)
        tmp = x + (x / 32);
        x5 = (tmp / 4) + (x / 16);
end

function x6 =  pmul_3_2(x)
        tmp = x + (x / 32) ;
        x6 = tmp - (tmp / 4);
end
```

Appendix B

Handwritten VHDL code for JPEG Encoder

```
--encoder.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY encoder IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic;
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END encoder;

ARCHITECTURE rtl OF encoder IS

component dct_top IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic;
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END component;

component rightshift IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic:='0';
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END component;

component qtz_zz_top IS
```

```
        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in    : in  signed(31 downto 0);
                    start_out  : out  std_logic;
                    data_out   : out signed(31 downto 0):=(others=>'0'));
END component;

component RLE IS

        PORT(
                         clk,reset,start_in : in  std_logic;
                         data_in    : in  signed(31 downto 0);
                         out_send   : out std_logic;
                         data_out   : out signed(31 downto 0):=(others=>'0'));
END component;

signal temp,temp1,temp2,temp3 : signed(31 downto 0);
signal data_in1 : signed(31 downto 0);
signal start,start1,start2,start3 : std_logic := '0';
signal st1 : std_logic := '0';

begin
           data_in1<=shift_left(data_in,7);
           st1<=st;
           DCT_1D_1: dct_top port map(clk,reset,st1,data_in1,start,temp);
           DCT_1D_2: dct_top port map(clk,reset,start,temp,start1,temp1);
           right_sh: rightshift port map(clk,reset,start1,temp1,start2,temp2);
           QTZ_zz   : qtz_zz_top port map(clk,reset,start2,temp2,start3,temp3);
           RLE_1    : RLE  port map(clk,reset,start3,temp3,start_out,data_out);
END rtl;


--dct_top.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY dct_top IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in    : in  signed(31 downto 0);
                    start_out  : out  std_logic;
                    data_out   : out signed(31 downto 0):=(others=>'0'));
END dct_top;

ARCHITECTURE rtl OF dct_top IS

component stage1 IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in    : in  signed(31 downto 0);
                    start_out  : out  std_logic:='0';
                    data_out0  : out signed(31 downto 0):=(others=>'0');
                    data_out1  : out signed(31 downto 0):=(others=>'0');
```

```
                data_out2  : out signed(31 downto 0):=(others=>'0');
                data_out3  : out signed(31 downto 0):=(others=>'0');
                data_out4  : out signed(31 downto 0):=(others=>'0');
                data_out5  : out signed(31 downto 0):=(others=>'0');
                data_out6  : out signed(31 downto 0):=(others=>'0');
                data_out7  : out signed(31 downto 0):=(others=>'0'));
END component;

component stage2 IS

        PORT(
                clk,reset,start_in : in  std_logic;
                data_in0   : in  signed(31 downto 0);
                data_in1   : in  signed(31 downto 0);
                data_in2   : in  signed(31 downto 0);
                data_in3   : in  signed(31 downto 0);
                data_in4   : in  signed(31 downto 0);
                data_in5   : in  signed(31 downto 0);
                data_in6   : in  signed(31 downto 0);
                data_in7   : in  signed(31 downto 0);
                start_out  : out  std_logic:='0';
                data_out0  : out signed(31 downto 0):=(others=>'0');
                data_out1  : out signed(31 downto 0):=(others=>'0');
                data_out2  : out signed(31 downto 0):=(others=>'0');
                data_out3  : out signed(31 downto 0):=(others=>'0');
                data_out4  : out signed(31 downto 0):=(others=>'0');
                data_out5  : out signed(31 downto 0):=(others=>'0');
                data_out6  : out signed(31 downto 0):=(others=>'0');
                data_out7  : out signed(31 downto 0):=(others=>'0'));
END component;

component stage3 IS

        PORT(
                clk,reset,start_in : in  std_logic;
                data_in0   : in  signed(31 downto 0);
                data_in1   : in  signed(31 downto 0);
                data_in2   : in  signed(31 downto 0);
                data_in3   : in  signed(31 downto 0);
                data_in4   : in  signed(31 downto 0);
                data_in5   : in  signed(31 downto 0);
                data_in6   : in  signed(31 downto 0);
                data_in7   : in  signed(31 downto 0);
                start_out  : out  std_logic:='0';
                data_out0  : out signed(31 downto 0):=(others=>'0');
                data_out1  : out signed(31 downto 0):=(others=>'0');
                data_out2  : out signed(31 downto 0):=(others=>'0');
                data_out3  : out signed(31 downto 0):=(others=>'0');
                data_out4  : out signed(31 downto 0):=(others=>'0');
                data_out5  : out signed(31 downto 0):=(others=>'0');
                data_out6  : out signed(31 downto 0):=(others=>'0');
                data_out7  : out signed(31 downto 0):=(others=>'0'));
END component;

component ptos IS

        PORT(
```

```
                clk,reset,start_in : in  std_logic;
                data_in0   : in  signed(31 downto 0);
                data_in1   : in  signed(31 downto 0);
                data_in2   : in  signed(31 downto 0);
                data_in3   : in  signed(31 downto 0);
                data_in4   : in  signed(31 downto 0);
                data_in5   : in  signed(31 downto 0);
                data_in6   : in  signed(31 downto 0);
                data_in7   : in  signed(31 downto 0);
                start_out  : out  std_logic:='0';
                data_out   : out signed(31 downto 0):=(others=>'0'));
END component;

component transpose IS

        PORT(
                clk,reset,start_in : in  std_logic;
                data_in   : in  signed(31 downto 0);
                start_out : out  std_logic:='0';
                data_out  : out signed(31 downto 0):=(others=>'0'));
END component;

signal temp,temp1,temp2:signed(31 downto 0);
signal s1_data_out0,s1_data_out1,s1_data_out2,s1_data_out3,s1_data_out4,s1_data_out5,
       s1_data_out6,s1_data_out7,s2_data_out0,s2_data_out1,s2_data_out2,s2_data_out3,
       s2_data_out4,s2_data_out5,s2_data_out6,s2_data_out7,s3_data_out0,s3_data_out1,
       s3_data_out2,s3_data_out3,s3_data_out4,s3_data_out5,s3_data_out6,s3_data_out7,
       ptos_data_out : signed(31 downto 0);
signal start,start1,start2,start3 : std_logic := '0';
signal st1 : std_logic := '0';

begin

    E1: stage1 port map(clk,reset,start_in,data_in,start,s1_data_out0,s1_data_out1,
                        s1_data_out2,s1_data_out3,s1_data_out4,s1_data_out5,s1_data_out6,
                        s1_data_out7);
    E2: stage2 port map(clk,reset,start,s1_data_out0,s1_data_out1,s1_data_out2,
                        s1_data_out3,s1_data_out4,s1_data_out5,s1_data_out6,s1_data_out7,
                        start1,s2_data_out0,s2_data_out1,s2_data_out2,s2_data_out3,
                        s2_data_out4,s2_data_out5,s2_data_out6,s2_data_out7);
    E3: stage3 port map(clk,reset,start1,s2_data_out0,s2_data_out1,s2_data_out2,
                        s2_data_out3,s2_data_out4,s2_data_out5,s2_data_out6,s2_data_out7,
                        start2,s3_data_out0,s3_data_out1,s3_data_out2,s3_data_out3,
                        s3_data_out4,s3_data_out5,s3_data_out6,s3_data_out7);
    E4: ptos port map  (clk,reset,start2,s3_data_out0,s3_data_out1,s3_data_out2,
                        s3_data_out3,s3_data_out4,s3_data_out5,s3_data_out6,s3_data_out7,
                        start3,ptos_data_out);
    E5:transpose port map(clk,reset,start3,ptos_data_out,start_out,data_out);
END rtl;

--stage1.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY stage1 IS
```

```
        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in    : in  signed(31 downto 0);
                    start_out  : out  std_logic:='0';
                    data_out0  : out signed(31 downto 0):=(others=>'0');
                    data_out1  : out signed(31 downto 0):=(others=>'0');
                    data_out2  : out signed(31 downto 0):=(others=>'0');
                    data_out3  : out signed(31 downto 0):=(others=>'0');
                    data_out4  : out signed(31 downto 0):=(others=>'0');
                    data_out5  : out signed(31 downto 0):=(others=>'0');
                    data_out6  : out signed(31 downto 0):=(others=>'0');
                    data_out7  : out signed(31 downto 0):=(others=>'0'));
END stage1;

ARCHITECTURE rtl OF stage1 IS

BEGIN
  process(clk,reset)
     variable compt : integer :=0;
     variable start : std_logic:='0';
     type mem is array(0 to 7) of signed(31 downto 0);
     variable input : mem := ((others=> (others=>'0')));
     begin
       if (clk'event and clk='1') then
         if(reset='0')then
            start:='0';
            compt:=0;
            input:=((others=> (others=>'0')));
            start_out <='0';
         else
           if(start_in='1')then
             input(compt):=data_in;
             if(compt=7)then
                compt:=0;
                start:='1';
             else
                compt:=compt+1;
                start:='0';
                start_out<='0';
             end if;
             if(start='1')then
                data_out0 <= input(0) + input(7);
                data_out1 <= input(0) - input(7);
                data_out4 <= input(1) + input(6);
                data_out5 <= input(1) - input(6);
                data_out2 <= input(2) + input(5);
                data_out3 <= input(2) - input(5);
                data_out6 <= input(3) + input(4);
                data_out7 <= input(3) - input(4);
                start_out<='1';
              end if;
           else
              start_out<='0';
           end if;
       end if;
     end if;
```

```
end process;

END rtl;

--stage2.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY stage2 IS

        PORT(
                  clk,reset,start_in : in  std_logic;
                  data_in0   : in  signed(31 downto 0);
                  data_in1   : in  signed(31 downto 0);
                  data_in2   : in  signed(31 downto 0);
                  data_in3   : in  signed(31 downto 0);
                  data_in4   : in  signed(31 downto 0);
                  data_in5   : in  signed(31 downto 0);
                  data_in6   : in  signed(31 downto 0);
                  data_in7   : in  signed(31 downto 0);
                  start_out  : out  std_logic:='0';
                  data_out0  : out signed(31 downto 0):=(others=>'0');
                  data_out1  : out signed(31 downto 0):=(others=>'0');
                  data_out2  : out signed(31 downto 0):=(others=>'0');
                  data_out3  : out signed(31 downto 0):=(others=>'0');
             data_out4  : out signed(31 downto 0):=(others=>'0');
                  data_out5  : out signed(31 downto 0):=(others=>'0');
                  data_out6  : out signed(31 downto 0):=(others=>'0');
                  data_out7  : out signed(31 downto 0):=(others=>'0'));
END stage2;

ARCHITECTURE rtl OF stage2 IS
function pmul_1_2(X : signed) return signed is
    variable pmul_1_2_tmp1_1 : signed(31 downto 0);
    variable pmul_1_2_tmp2_1 : signed(31 downto 0);
  begin
    pmul_1_2_tmp1_1 := shift_right(X,3) - shift_right(X,7);
    pmul_1_2_tmp2_1 := pmul_1_2_tmp1_1 - shift_right(X,11);
    return pmul_1_2_tmp1_1 + shift_right(pmul_1_2_tmp2_1,1);
  end pmul_1_2;

  function pmul_1_1(X : signed) return signed is
  begin
    return X - shift_right(X,3) - shift_right(X,7);
  end pmul_1_1;
BEGIN
        process(clk,reset)
    variable x0,x1,x2,x3,x4,x5,x6,x7,xa,xb:signed(31 downto 0);
    begin
    if (clk'event and clk='1') then
      if(reset='0')then
         start_out <='0';
      else
         if(start_in='1')then
            x0 := data_in0;
```

```
            x1 := data_in1;
            x2 := data_in2;
            x3 := data_in3;
            x4 := data_in4;
            x5 := data_in5;
            x6 := data_in6;
            x7 := data_in7;
            xa := pmul_1_2(x3);
            x3 := pmul_1_1(x3);
            xb := pmul_1_2(x5);
            x5 := pmul_1_1(x5);
            x3 := x3 + xb;
            x5 := x5 - xa;
            xa := x0 + x6;
            x6 := x0 - x6;
            xb := x4 + x2;
            x2 := x4 - x2;
            x0 := xa + xb;
            x4 := xa - xb;
            data_out0 <= x0;
            data_out1 <= x1;
            data_out2 <= x2;
            data_out3 <= x3;
            data_out4 <= x4;
            data_out5 <= x5;
            data_out6 <= x6;
            data_out7 <= x7;
            start_out<='1';
         else
            start_out<='0';
         end if;
      end if;
   end if;
end process;

END rtl;

--stage3.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;


ENTITY stage3 IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in0   : in  signed(31 downto 0);
                    data_in1   : in  signed(31 downto 0);
                    data_in2   : in  signed(31 downto 0);
                    data_in3   : in  signed(31 downto 0);
                    data_in4   : in  signed(31 downto 0);
                    data_in5   : in  signed(31 downto 0);
                    data_in6   : in  signed(31 downto 0);
                    data_in7   : in  signed(31 downto 0);
                    start_out  : out  std_logic:='0';
```

```
                    data_out0  : out signed(31 downto 0):=(others=>'0');
                    data_out1  : out signed(31 downto 0):=(others=>'0');
                    data_out2  : out signed(31 downto 0):=(others=>'0');
                    data_out3  : out signed(31 downto 0):=(others=>'0');
                    data_out4  : out signed(31 downto 0):=(others=>'0');
                    data_out5  : out signed(31 downto 0):=(others=>'0');
                    data_out6  : out signed(31 downto 0):=(others=>'0');
                    data_out7  : out signed(31 downto 0):=(others=>'0'));
END stage3;

ARCHITECTURE rtl OF stage3 IS
 function pmul_2_2(X : signed) return signed is
  begin
    return shift_right(X,1);
  end pmul_2_2;

  function pmul_2_1(X : signed) return signed is
    variable pmul_2_1_tmp_1 : signed(31 downto 0);
  begin
    pmul_2_1_tmp_1 := shift_right(X,9) - X;
    return shift_right(pmul_2_1_tmp_1,2) - pmul_2_1_tmp_1;
  end pmul_2_1;

  function pmul_3_2(X : signed) return signed is
    variable pmul_3_2_tmp_1 : signed(31 downto 0);
  begin
    pmul_3_2_tmp_1 := X + shift_right(X,5);
    return pmul_3_2_tmp_1 - shift_right(pmul_3_2_tmp_1,2);
  end pmul_3_2;

  function pmul_3_1(X : signed) return signed is
    variable pmul_3_1_tmp_1 : signed(31 downto 0);
  begin
    pmul_3_1_tmp_1 := X + shift_right(X,5);
    return shift_right(pmul_3_1_tmp_1,2) + shift_right(X,4);
  end pmul_3_1;
BEGIN
process(clk,reset)
    variable x0,x1,x2,x3,x4,x5,x6,x7,xa,xb:signed(31 downto 0);

    begin
      if (clk'event and clk='1') then
        if(reset='0')then
           start_out <='0';
        else
           if(start_in='1')then
               x0 := data_in0;
               x1 := data_in1;
               x2 := data_in2;
               x3 := data_in3;
               x4 := data_in4;
               x5 := data_in5;
               x6 := data_in6;
               x7 := data_in7;

               xa := pmul_2_2(x1);
               x1 := pmul_2_1(x1);
```

```
                xb := pmul_2_2(x7);
                x7 := pmul_2_1(x7);
                x1 := x1 - xb;
                x7 := x7 + xa;
                xa := x1 + x3;
                x3 := x1 - x3;
                xb := x7 + x5;
                x5 := x7 - x5;
                x1 := xa + xb;
                x7 := xa - xb;

                xa := pmul_3_2(x2);
                x2 := pmul_3_1(x2);
                xb := pmul_3_2(x6);
                x6 := pmul_3_1(x6);
                x2 := xb + x2;
                x6 := x6 - xa;

                data_out0 <= x0;
                data_out1 <= x1;
                data_out2 <= x2;
                data_out3 <= x3;
                data_out4 <= x4;
                data_out5 <= x5;
                data_out6 <= x6;
                data_out7 <= x7;

                start_out<='1';
            else
                start_out<='0';
            end if;
        end if;
    end if;
end process;

END rtl;

--ptos.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;


ENTITY ptos IS

        PORT(
                clk,reset,start_in : in  std_logic;
                data_in0   : in  signed(31 downto 0);
                data_in1   : in  signed(31 downto 0);
                data_in2   : in  signed(31 downto 0);
                data_in3   : in  signed(31 downto 0);
                data_in4   : in  signed(31 downto 0);
                data_in5   : in  signed(31 downto 0);
                data_in6   : in  signed(31 downto 0);
                data_in7   : in  signed(31 downto 0);
                start_out  : out  std_logic:='0';
```

```
                  data_out   : out signed(31 downto 0):=(others=>'0'));
END ptos;

ARCHITECTURE rtl OF ptos IS
  type t_state is (s0,s1);
  signal state: t_state;
BEGIN
process(clk,reset)
   variable compt : integer :=0;
   type mem is array(0 to 7) of signed(31 downto 0);
   variable input : mem := ((others=> (others=>'0')));

   begin
     if (clk'event and clk='1') then
       if(reset='0')then
         state<=s0;
         compt:=0;
         input:=((others=> (others=>'0')));
         start_out <='0';
       else
         case state is
           when s0=>
             if(start_in='1')then
                input(0) := data_in0;
                input(1) := data_in1;
                input(4) := data_in4;
                input(5) := data_in5;
                input(2) := data_in2;
                input(3) := data_in3;
                input(6) := data_in6;
                input(7) := data_in7;
                compt:=0;
                start_out<='0';
                state<=s1;
             else
                start_out<='0';
             end if;
          when s1=>
             data_out <= input(compt);
             start_out<='1';
             if(compt=7)then
                compt:=0;
                state<=s0;
                start_out<='0';
             else
                compt:=compt+1;
             end if;
          end case;
       end if;
    end if;
end process;

END rtl;

--transpose.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;


ENTITY transpose IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic:='0';
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END transpose;

ARCHITECTURE rtl OF transpose IS
  type t_state is (s0,s1);
  signal state : t_state;
BEGIN
process(clk,reset)
   variable x0,x1,x2,x3,x4,x5,x6,x7,xa,xb : signed(31 downto 0);
   type matrix is array(0 to 63) of signed(31 downto 0);
   variable m1 : matrix := ((others=> (others=>'0')));
   variable j,k,l : std_logic_vector(7 downto 0);
   variable compt,compt2 : integer := 0;

   begin
     if (clk'event and clk='1') then
       if(reset='0')then
          start_out <='0';
          l := "00000000";
          compt:=0;
       else
          case state is
            when s0=>
               if(start_in='1')then
                  m1(compt) := data_in;
                  if(compt=63)then
                     compt:=0;
                     state<=s1;
                  else
                     compt:=compt+1;
                  end if;
               else
                  start_out<='0';
               end if;

            when s1=>
                data_out<=m1(conv_integer(l));
                start_out<='1';
                if(compt2=63)then
                   compt2:=0;
                   state<=s0;
                else
                   if(k="00000111")then
                      k:="00000000";
                      j:=j+1;
                      l:=j;
                   else
```

```
                    k:=k+1;
                    l:=l+"00001000";
                  end if;
                  compt2:=compt2+1;
                end if;
            end case;

      end if;
    end if;
end process;

END rtl;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

--rightshift.vhdl file
ENTITY rightshift IS

        PORT(
                clk,reset,start_in : in  std_logic;
                data_in   : in  signed(31 downto 0);
                start_out : out  std_logic:='0';
                data_out  : out signed(31 downto 0):=(others=>'0'));
END rightshift;

ARCHITECTURE rtl OF rightshift IS

BEGIN
  process(clk,reset)
    type t_s is array (0 to 63) of integer;
    variable s : t_s := (1024, 1138, 1730, 1609, 1024, 1609, 1730, 1138,
    1138, 1264, 1922, 1788, 1138, 1788, 1922, 1264, 1730,1922,2923, 2718,
    1730, 2718, 2923, 1922, 1609, 1788, 2718, 2528, 1609,2528,2718, 1788,
    1024, 1138, 1730, 1609, 1024, 1609, 1730, 1138, 1609,1788,2718, 2528,
    1609, 2528, 2718, 1788, 1730, 1922, 2923, 2718, 1730,2718,2923, 1922,
    1138, 1264, 1922, 1788, 1138, 1788, 1922, 1264);
    variable i : integer :=0;
    variable temp : signed(31 downto 0);
  begin
   if(reset='0')then
    i:=0;
    data_out <=(others=>'0');
    start_out <='0';
   else
    if (clk'event and clk='1') then
     if(start_in='1')then
       temp:= 524287 - shift_right(data_in,31);
       data_out<=resize(shift_right((data_in * s(i) + temp),20),32);
       start_out <='1';
         if(i=63)then
           i:=0;
         else
           i:=i+1;
         end if;
```

```
      else
        start_out <='0';
    end if;
   end if;
  end if;
end process;

END rtl;

--qtz_zz_top.file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;


ENTITY qtz_zz_top IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic;
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END qtz_zz_top;

ARCHITECTURE rtl OF qtz_zz_top IS

component qtz IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic:='0';
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END component;

component zigzag IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic:='0';
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END component;

signal temp:signed(31 downto 0);
signal start:std_logic:='0';

begin
   QTZ_1: qtz port map(clk,reset,st,data_in,start,temp);
   ZZ_2 : zigzag port map(clk,reset,start,temp,start_out,data_out);

END rtl;

--qtz.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY qtz IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic:='0';
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END qtz;

ARCHITECTURE rtl OF qtz IS

   type mem is array(0 to 63) of signed(31 downto 0);
   signal input,input1 : mem := ((others=> (others=>'0')));
   attribute ramstyle : string;
   attribute ramstyle of input,input1 : signal is "M4K";
   BEGIN
   process(clk,reset)
     variable compt,compt1,compt2: integer :=0;

     type memory_type_QT is array (0 to 63) of integer;
     constant QT : memory_type_QT := (16, 11, 10 ,16 ,24 ,40 ,51 ,61 ,12 ,12 ,14 ,19 ,
     26 ,58 ,60, 55 ,14 ,13 ,16 ,24 ,40 ,57, 69 ,56 ,14 ,17 ,22 ,29, 51 ,87 ,80 , 62 ,
     18 ,22 ,37 , 56 ,68 ,109 ,103, 77 ,24 ,35 ,55 ,64 ,81 ,104 ,113 ,92 ,49 ,64 ,78 ,
     87 ,103 ,121 ,120 ,;101 ,72 ,92 ,95 ,98 ,112 ,100 ,103 ,99);
     type memory_type_zigzag is array (0 to 63) of integer;
     constant zigzag : memory_type_zigzag := (0, 1, 5, 6, 14, 15, 27, 28, 2, 4, 7, 13,
     16, 26, 29, 42, 3, 8, 12, 17, 25, 30, 41, 43, 9, 11, 18, 24, 31, 40,44,53,10, 19,
     23, 32, 39, 45, 52, 54, 20, 22, 33, 38, 46, 51, 55, 60, 21, 34, 37, 47,50,56, 59,
     61, 35, 36, 48, 49, 57, 58, 62,63);
     type memory_type_DivTab is array (0 to 120) of integer;
     constant DivTab : memory_type_DivTab := (65536, 32768, 21845, 16384, 13107,
     10923, 9362, 8192, 7282, 6554, 5958, 5461, 5041, 4681, 4369, 4096, 3855, 3641,
     3449, 3277, 3121, 2979, 2849, 2731, 2621, 2521, 2427, 2341, 2260, 2185, 2114,
     2048, 1986, 1928, 1872, 1820, 1771, 1725, 1680, 1638, 1598, 1560, 1524, 1489,
     1456, 1425, 1394, 1365, 1337, 1311, 1285, 1260, 1237, 1214, 1192, 1170, 1150,
     1130, 1111, 1092, 1074, 1057, 1040, 1024, 1008, 993, 978, 964, 950, 936,923,
     910, 898,886, 874, 862, 851, 840, 830, 819, 809, 799, 790, 780, 771, 762,753,
     745, 736, 728, 720, 712,705, 697, 690, 683,676, 669, 662, 655, 649, 643, 636,
     630, 624, 618, 612, 607, 601, 596, 590,585, 580,575, 570, 565, 560, 555, 551,
     546, 542);
     variable temp : integer;

     begin

     if (clk'event and clk='1') then
         if(reset='0')then
            compt:=0;
         else
            if(start_in='1')then
               temp :=DivTab(QT(compt)-1);
               if(data_in<0)then
                  data_out <= resize(shift_right(data_in * (-1)* temp,16) * (-1),32);
               else
                  data_out <= resize(shift_right(data_in * temp,16),32);
```

```
                end if;
                if(compt=63)then
                   compt:=0;
                else
                   compt := compt + 1;
                end if;
                start_out<='1';
            else
                start_out<='0';
            end if;
       end if;
    end if;
end process;

END rtl;

--zigzag.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY zigzag IS

        PORT(
                    clk,reset,start_in : in  std_logic;
                    data_in   : in  signed(31 downto 0);
                    start_out : out  std_logic:='0';
                    data_out  : out signed(31 downto 0):=(others=>'0'));
END zigzag;

ARCHITECTURE rtl OF zigzag IS
  type state is (s0,s1);
  signal s:state;
  type mem is array(0 to 63) of signed(31 downto 0);
  signal input,input1 : mem := ((others=> (others=>'0')));
  attribute ramstyle : string;
  attribute ramstyle of input,input1 : signal is "M4K";
  BEGIN
  process(clk,reset)
    variable compt,compt1,compt2: integer :=0;
    type memory_type_zigzag is array (0 to 63) of integer;
    constant zigzag : memory_type_zigzag := (0, 1, 5, 6, 14, 15, 27, 28, 2, 4, 7, 13,
    16, 26, 29, 42, 3, 8, 12, 17, 25, 30, 41, 43, 9, 11, 18, 24, 31, 40, 44, 53, 10,
    19, 23, 32, 39, 45, 52, 54, 20, 22, 33, 38, 46, 51, 55, 60, 21, 34, 37, 47, 50,
    56, 59, 61, 35, 36, 48, 49, 57, 58, 62,63);
    begin

      if (clk'event and clk='1') then
          if(reset='0')then
             s <= s0;
             compt:=0;
             compt1:=0;
          else
             case s is
               when s0 =>
                  if(start_in='1')then
```

```
                    input(zigzag(compt)) <= data_in;
                    if(compt=63)then
                       compt:=0;
                       s<=s1;
                    else
                       compt := compt + 1;
                       s<=s0;
                    end if;
                 end if;
              when s1 =>
                  start_out <='1';
                  data_out <= input(compt1);
                  if(compt1=63)then
                     compt1:=0;
                     s<=s0;
                  else
                     compt1 := compt1 + 1;
                     s<=s1;
                  end if;
            end case;
         end if;
    end if;
end process;

END rtl;

--rle.vhdl file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;


ENTITY RLE IS

        PORT(
                  clk,reset,start_in : in  std_logic;
                  data_in   : in  signed(31 downto 0);
                  out_send  : out std_logic;
                  data_out  : out signed(31 downto 0):=(others=>'0'));
END RLE;

ARCHITECTURE rtl OF RLE IS
   type mem is array(0 to 63) of signed(31 downto 0);
   signal input : mem := ((others=> (others=>'0')));
   type mem1 is array(0 to 127) of signed(31 downto 0);
   signal output : mem1 := ((others=> (others=>'0')));
   type state is (s0,s01,s1,s11,s12,s13,s2,s21,s22,s23,s3,s31,s4);
   signal st:state;
   BEGIN
         process(clk,reset)

       variable compt,k,i: integer :=0;
       variable temp: signed(31 downto 0);
       variable EOB_idx,ZRL: integer :=0;
       variable rb_done,done,zero: boolean ;
```

```
begin

  if (clk'event and clk='1') then
    if(reset='0')then
       st <= s0;
       compt:=1;
       i := 0;
       k := 0;
       EOB_idx := 0;
       out_send <='0';
     else
       case st is
         when s0=>
           if(start_in='1')then
              input(compt)<=data_in;
              out_send <='0';
              if(compt=63)then
                 compt:=0;
                 i := 63;
                 k := 1;
                 EOB_idx := 63;
                 st <= s01;
              else
                 compt:= compt+1;
                 st <= s0;
              end if;
           end if;
        when s01=>
           done:=(input(i)=0);
           st <= s1;
        when s1=>
           if(done)then
             st<=s11;
             i := i-1;
           else
             st<=s12;
             EOB_idx := i;
             i := 1;
           end if;
        when s11=>
           done := (input(i)=0);
           st<=s1;
        when s12=>
           output(k)<=input(i);
           i:=i+1;
           k:=k+1;
           st<=s13;
        when s13=>
           output(k)<= to_signed(0,32);
           k:=k+1;
           st<=s2;
           zero :=(input(i)=0);
        when s2=>
           if(i=EOB_idx)then
              output(k)<=input(i);
              st<=s23;
              i:=1;
```

```
                else
                   if(zero)then--input(i)=0
                      ZRL := ZRL + 1;
                      i := i + 1;
                      st<=s22;
                   else
                      output(k)<=input(i);
                      i := i + 1;
                      k := k + 1;
                      st<=s21;
                   end if;
                end if;
             when s21=>
                output(k)<=to_signed(ZRL,32);
                ZRL := 0;
                k := k + 1;
                st<=s2;
                zero :=(input(i)=0);
             when s22=>
                zero :=(input(i)=0);
                st<=s2;
             when s23=>
                k:=k+1;
                output(k)<=to_signed(ZRL,32);
                st<=s3;
             when s3=>
                if (EOB_idx < 64)then
                    k := k + 1;
                    output(k)<=to_signed(0,32);
                    st <= s31;
                 else
                    st <= s4;
                 end if;
             when s31=>
                 k := k + 1;
                 output(k)<=to_signed(0,32);
                 st <= s4;
             when s4=>
                 if(i=k)then
                    data_out<=output(i);
                    out_send <='1';
                    i:=1;
                    k:=1;
                    st <= s0;
                 else
                    data_out<=output(i);
                    out_send <='1';
                    i := i + 1;
                    st <= s4;
                 end if;
           end case;
       end if;
    end if;
end process;

END rtl;
```

Bibliography

[1] *Cyclone II Device Handbook, Volume 1*, Altera Corporation, Feburary 2007.

[2] Altera stratix ii dsp blocks. [Online]. Available: http://www.altera.com/devices/fpga/stratix-fpgas/stratix-ii/stratix-ii/features/dsp/st2-dsp_block.html

[3] J. Serot., *Caph language reference manual - v 1.5*. [Online]. Available: http://wwwlasmea.univ- bpclermont.fr/Personnel/Jocelyn.Serot/caph.html

[4] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[5] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel Processing in a Restructurable Computer System," *Electronic Computers, IEEE Transactions on*, pp. 747–755, Dec. 2006. [Online]. Available: http://dx.doi.org/10.1109/PGEC.1963.263558

[6] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, Eds., *Splash 2: FPGAs in a Custom Computing Machine*. Wiley-IEEE Computer Society Press, 1996.

[7] W. Carter, K. Duong, R. H. Freeman, H. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze, "A user programmable reconfiguration gate array," in *Proceedings of IEEE Custom Integrated Circuits Conference*, May 1986, pp. 233–235.

[8] *Stratix Device Handbook, Volume 1*, Altera, July 2005.

[9] *Stratix II device handbook, Volume 1*, Altera Corporation, May 2007.

[10] *Virtex-6 FPGA Configurable Logic Block User Guide*, Xilinx, February 2012.

[11] Altera stratix ii architecture. [Online]. Available: http://www.altera.com/devices/fpga/stratix-fpgas/stratix-ii/stratix-ii/st2-index.jsp

[12] *Institute of Electrical and Electronics Engineers (IEEE) : 1076-2000 IEEE Standard VHDL Language Reference Manual (2000).*

[13] *Institute of Electrical and Electronics Engineers(IEEE) : 1364-2001 IEEE Standard Verilog Hardware Description Language (2001).*

[14] *International Organization for Standardization (ISO) : ISO/IEC 9899 : 1999.*

[15] *Institute of Electrical and Electronics Engineers (IEEE) : 16662005 IEEE standard SystemC.*

[16] Impulse-codeveloper. [Online]. Available: http://www.impulsec.com/

[17] Handle-c. [Online]. Available: http://www.celoxica.com/

[18] Mitrion-c. [Online]. Available: http://www.mitrionics.com/

[19] D. S. Poznanovic, "Application development on the src computers, inc. systems," in *IPDPS*, 2005.

[20] M. Gokhale, J. M. Stone, J. M. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *FCCM*, 2000, pp. 49–58.

[21] T. Hoare, *Communicating Sequential Processes.* Prentice Hall International, 1985.

[22] W. Najjar, W. Bohm, B. Draper, R. Hammes, J.and Rinker, R. Beveridge, M. Chawathe, and C. Ross, "High-level language abstraction for reconfigurable computing," *IEEE Computer*, vol. 36, no. 8, pp. 63–69, August 2003.

[23] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark : A high-lev l synthesis framework for applying parallelizing compiler transformations," *VLSI Design, International Conference on*, 2003.

[24] Y. D. Yankova, K. Bertels, S. Vassiliadis, G. Kuzmanov, and R. Chaves, "Hll-to-hdl generation: Results and challenges," in *Proceedings of ProRisc 2006*, November 2006.

[25] Z. ul Abdin and B. Svensson, "A study of design efficiency with a high-level language for fpgas," in *IPDPS*, 2007, pp. 1–7.

[26] Y. Yankova, K. Kuzmanov, G.and Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator," in *International Conference on Field Programmable Logic and Applications*, 2007.

[27] W. A. Najjar, E. A. Leeb, and G. Gaoc, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, pp. 1907–1929, December 1999.

[28] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing kansas lava," in *21st International Symposium on Implementation and Application of Functional Languages*, LNCS 6041. LNCS 6041, 11/2009 2009. [Online]. Available: http://www.ittc.ku.edu/csdl/fpg/sites/default/files/kansas-lava-ifl09.pdf

[29] J. Agron, "Domain-specific language for hw/sw co-design for fpgas," in *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, ser. DSL '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 262–284. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03034-5_13

[30] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium, Proceedings Colloque sur la Programmation.* London, UK, UK: Springer-Verlag, 1974, pp. 362–376. [Online]. Available: http://dl.acm.org/citation.cfm?id=647323.721501

[31] J. W. Backus, "Is computer science based on the wrong fundamental concept of program? an extended concept," in *Algorithmic Languages*, de Bakker/van Vliet, Ed. North Holland, 1981, p. 133.

[32] S. Wail and D. Abramson, "Can data-flow machines be programmed with an imperative language?" in *Advanced Topics in Dataflow Computing*, B. Editors Gao and Gaudiot, Eds. IEEE Computer Society Press, 1995, pp. 229–266.

[33] Arvind, D. Culler, and G. Maa, "Assessing the benefits of fine-grain parallelism in dataflow programs," in *Supercomputing '88*, vol. 1, November 1988, pp. 60–69.

[34] J. Backus, "Can programming be liberated from the von neumann style?: a functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359579

[35] K.-S. Weng, "Stream oriented computation recursive data flow schemas," Laboratory for Computer Science, MIT,Cambridge, Tech. Rep., 1975.

[36] P. R. Kosinski, "A data flow language for operating systems programming," in *Proceeding of ACM SIGPLAN - SIGOPS interface meeting on Programming languages - operating systems*. New York, NY, USA: ACM, 1973, pp. 89–94. [Online]. Available: http://doi.acm.org/10.1145/800021.808289

[37] P. Whiting and R. Pascoe, "A history of data-flow languages," *IEEE Annals of the History of Computing*, vol. 16, no. 4, pp. 38 – 59, 1994.

[38] A. L. Davis and R. M. Keller, "Data flow program graphs," *IEEE Computer*, vol. 15, no. 2, pp. 26–41, 1982.

[39] W. M. Johnston, J. R. P. Hanna, and R. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1 – 34, March 2004.

[40] E. Lee and D. Messerschmitt, "Synchronous data flow," in *Proceedings of the IEEE*, vol. 75, no. 9, Sept. 1987, pp. 1235 – 1245.

[41] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," vol. 44, no. 2, pp. 397–408, 1996. [Online]. Available: http://cat.inist.fr/?aModele=afficheN&cpsidt=3024849

[42] T. M. Parks, J. L. Pino, and E. A. Lee, "A comparison of synchronous and cycle-static dataflow," in *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, ser. ASILOMAR '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 204–. [Online]. Available: http://dl.acm.org/citation.cfm?id=784104.784293

[43] P. K. Murthy and E. A. Lee, "Multidimensional synchronous dataflow," *IEEE Transactions on Signal Processing*, vol. 50, pp. 3306–3309, 2002.

[44] S. Neuendorffer and K. Vissers, "Streaming systems in fpgas," *Embedded Computer Systems: Architectures, Modeling, and Simulation (Lecture Notes in Computer Science)*, vol. 5114/2008, pp. 147–156, 2008.

[45] O. Gelly, "Lau software systems: a high level data driven language for parallel processing," in *International Conference on Parallel Processing*, 1976.

[46] E. A. Ashcroft and W. W. Wadge, "Lucid, a nonprocedural language with iteration," *Commun. ACM*, vol. 20, pp. 519–526, July 1977. [Online]. Available: http://doi.acm.org/10.1145/359636.359715

[47] W. W. Wadge and E. A. Ashcroft, *LUCID, the dataflow programming language*. San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[48] Arvind, K. Gostelow, and W. Plouffe, "An asynchronous programming language and computing machine," University of California, Irvine, Tech. Rep. TR 114a, 1978.

[49] J. Herath, N. Saito, K. Toda, Y. Yamaguchi, and T. Yuba, "Dcbl: dataflow computing based language with n-value logic," in *Proceedings of 1986 ACM Fall joint computer conference*, ser. ACM '86. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 353–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=324493.324586

[50] J. Mcgraw, S. Skedzielewski, S. Allan, O. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas, *SISAL: Streams and iteration in a single assignment language, language reference manual version 1.2*. Lawrence-Livermore-National-Laboratory, Mar. 1985.

[51] J. Glauert, "A single assignment language for dataflow computing," Master's thesis, University of Manchester, Manchester,U.K., 1978.

[52] J. Gurd, "The manchester dataflow machine," *Computer Physics Communications*, vol. 37, no. 1-3, pp. 49–62, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0010465585901353

[53] J. Eker and J. Janneck, "Cal language report," University of California at Berkeley, Tech. Rep. ERL Technical Memo UCB/ERL M03/48, December 2003.

[54] R. Thavot, R. Mosqueron, M. Alisafaee, C. Lucarz, M. Mattavelli, J. Dubois, and V. Noel, "Dataflow design of a co-processor architecture for image processing," in *Proceedings of the 2008 Conference on Design and Architectures for Signal and Image Processing*, 2008.

[55] W. D. Clinger, "Foundations of actor semantics," Cambridge, MA, USA, Tech. Rep., 1981.

[56] S. Bhattacharyya, G. Brebner, J. Eker, J. Janneck, M. Mattavelli, and C. Platen, "Opendf– a dataflow toolset for reconfigurable hardware and multicore systems," in *First Swedish workshop on multi-core computing(MCC)*, 2008.

[57] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the mpeg reconfigurable video coding framework," *J. Signal Process. Syst.*, vol. 63, no. 2, pp. 251–263, May 2011. [Online]. Available: http://dx.doi.org/10.1007/s11265-009-0399-3

[58] [Online]. Available: http://orcc.sourceforge.net/

[59] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: An mpeg-4 simple profile decoder case study," in *SiPS*, 2008, pp. 287–292.

[60] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow program: An mpeg-4 simple profile decoder case study," in *SiPS*, 2008, pp. 281–286.

[61] [Online]. Available: http://orcc.sourceforge.net/getting-started/code-generation/

[62] M. Wipliez, G. Roquier, and J.-F. Nezan, "Software code generation for the rvc-cal language," *J. Signal Process. Syst.*, vol. 63, no. 2, pp. 203–213, May 2011. [Online]. Available: http://dx.doi.org/10.1007/s11265-009-0390-z

[63] M. Wipliez, G. Roquier, M. Raulet, J.-F. Nezan, and O. Dforges, "Code generation for the mpeg reconfigurable video coding framework: From cal actions to c functions," in *ICME'08*, 2008, pp. 1049–1052.

[64] A. Dahlin, J. Ersfolk, G. Yang, and J. Lilius, "Configurable video decoding in a dataflow language," in *Conference on Design and Architectures for Signal and Image Processing (DASIP '09)*, 2009.

[65] A. Dahlin, J. Ersfolk, G. Yang, H. Habli, and J. Lilius, "The canals language and its compiler," in *12th International Workshop on Software and Compilers for Embedded Systems*, 2009.

[66] Streamit homepage. [Online]. Available: http://cag.csail.mit.edu/streamit

[67] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC*, 2002, pp. 179–196.

[68] W. Thies, "Language and compiler support for stream programs," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institue of Technology, February 2009.

[69] M. Hormati, A. andKudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: efficient realization of streaming applications on fpgas," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems(CASES '08)*, 2008.

[70] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal, "Baring it all to software: The raw machine," Cambridge, MA, USA, Tech. Rep., 1997.

[71] I. Buck, "Brook spec v0.2, technical report cstr 2003-04 10/31/03 12/5/03," Stanford University, Tech. Rep., 2003.

[72] Gpu brook source code. [Online]. Available: http://sourceforge.net/projects/brook/

[73] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi, "Merrimac: Supercomputing with streams," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050187

[74] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015800

[75] F. Plavec, Z. Vranesic, and S. Brown, "Towards compilation of streaming programs into fpga hardware," in *Forum on Specification, Verification and Design Languages(FDL 2008)*, 2008, pp. 67 – 72.

[76] F. Plavec, "Stream computing on fpgas," Ph.D. dissertation, Graduate Department of Electrical and Computer Engineering,University of Toronto, 2010.

[77] (November 2007) Nios ii c-to-hardware acceleration compiler. Altera. [Online]. Available: http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html

[78] *Quartus II Handbook*, Altera.

[79] J. Sérot, F. Berry, and S. Ahmed, "Implementing stream-processing applications on fpgas : a dsl-based approach," in *21st International Conference on Field Programmable Logic and Applications*, Crete, Sep 2011.

[80] S. North and E. Koutsofios, "Applications of graph visualization," in *Graphics Interface*, Banff, Alberta, 1994.

[81] Graphviz - graph visualization software. [Online]. Available: www.graphviz.org/

[82] J. Serot, "The semantics of a purely functional graph notation system," in *9th Symposium on Trends in Functional Programming*, 2008.

[83] J. Serot, F. Berry, and S. Ahmed, "Caph: A language for implementing stream-processing applications on fpgas," in *Embedded Systems Design with FPGAs*, P. Athanas, D. Pnevmatikatos, and N. Sklavos, Eds. Springer, 2012.

[84] P. Chalimbaud and F. Berry, "Versatile imaging architecture based on a system on chip," in *FPL*, 2004, pp. 1162–1164.

[85] *LUPA 4000: 4 MegaPixel CMOS Image Sensor*, Cypress Semiconductor Corporation, 2009.

[86] J. Leconte, "Areascan cameras: How to choose between global and rolling shutter," *ATMEL*, pp. 37 –39, 2006.

[87] K. Suzuki, I. Horiba, and N. Sugie, "Linear-time connected-component labeling based on sequential local operations," *Computer Vision and Image Understanding*, vol. 89, no. 1, pp. 1–23, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077314202000309

[88] T. Gotoh, Y. Ohta, M. Yoshida, and Y. Shirai, "High-speed algorithm for component labeling," *Systems and Computers in Japan*, vol. 21, no. 5, pp. 74–84, 1990. [Online]. Available: http://dx.doi.org/10.1002/scj.4690210507

[89] M. Komeichi, Y. Ohta, T. Gotoh, T. Mima, and M. Yoshida, "Video-rate labeling processor," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, ser. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, P. J. S. Hutzler and A. J. Oosterlinck, Eds., vol. 1027, 1989, p. 69.

[90] L. Thurfjell, E. Bengtsson, and B. Nordin, "A new three-dimensional connected components labeling algorithm with simultaneous object feature extraction capability," *CVGIP: Graph. Models Image Process.*, vol. 54, no. 4, pp. 357–364, Jul. 1992. [Online]. Available: http://dx.doi.org/10.1016/1049-9652(92)90083-A

[91] R. Lumia, "A new three-dimensional connected components algorithm," *Computer Vision, Graphics, and Image Processing*, vol. 23, no. 2, pp. 207–217, 1983. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0734189X83901135

[92] R. Lumia, L. Shapiro, and O. Zuniga, "A new connected components algorithm for virtual memory computers," *Computer Vision, Graphics, and Image Processing*, vol. 22, no. 2, pp. 287–300, 1983. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0734189X83900713

[93] A. Rosenfeld, "Connectivity in digital pictures," *J. ACM*, vol. 17, no. 1, pp. 146–160, Jan. 1970. [Online]. Available: http://doi.acm.org/10.1145/321556.321570

[94] M. B. Dillencourt, H. Samet, and M. Tamminen, "A general approach to connected-component labeling for arbitrary image representations," *J. ACM*, vol. 39, no. 2, pp. 253–280, Apr. 1992. [Online]. Available: http://doi.acm.org/10.1145/128749.128750

[95] P. Bhattacharya, "Connected component labeling for binary images on a reconfigurable mesh architecture," *J. Syst. Archit.*, vol. 42, no. 4, pp. 309–313, Nov. 1996. [Online]. Available: http://dx.doi.org/10.1016/1383-7621(96)00027-6

[96] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, no. 8, pp. 461–464, Aug. 1979. [Online]. Available: http://doi.acm.org/10.1145/359138.359141

[97] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *J. ACM*, vol. 13, no. 4, pp. 471–494, Oct. 1966. [Online]. Available: http://doi.acm.org/10.1145/321356.321357

[98] R. Haralick and L. Shapiro, *Computer and Robot Vision*. Addison Wesley, 1992, vol. Volume 1.

[99] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[100] L. di Stefano and A. Bulgarelli, "A simple and efficient connected components labeling algorithm," in *Proceedings of the 10th International Conference on Image Analysis and Processing*, ser. ICIAP '99. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=839281.840794

[101] R. Walczyk, A. Armitage, and D. Binnie, "Comparative study on connected component labeling algorithms for embedded video processing systems." in *IPCV'10*,

H. Deligiannidis, Ed. Las Vegas, USA: CSREA Press, 2010, vol. 2. [Online]. Available: http://researchrepository.napier.ac.uk/3901/

[102] R. Haralick, *Real-Time Parallel Computing Image Analysis*. New York: Plenum Press, 1981, ch. Some Neighborhood Operators, pp. 11–35.

[103] E. Mozef, S. Weber, J. Jaber, and G. Prieur, "Parallel architecture dedicated to image component labeling in o(n log n): Fpga implementation," pp. 120–125, 1996.

[104] F. Chang, C.-J. Chen, and C.-J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Computer Vision and Image Understanding*, vol. 93, no. 2, pp. 206–220, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.cviu.2003.09.002

[105] C. T. Johnston and D. G. Bailey, "Fpga implementation of a single pass connected components algorithm," in *DELTA*, 2008, pp. 228–231.

[106] D. G. Bailey and C. T. Johnston, "Single pass connected components analysis," *Image and Vision Computing New Zealand*, vol. 10, December 2007. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/21698715

[107] *Digital compression and coding of continuous-tone still images Requirements and guidelines*, International Telecommunication Union Std.

[108] N. Ahmed, T. Natarajan, and K. Rao, "Discrete cosine transform," in *IEEE Transactions on Computers*, vol. C-32, Jan. 1974, pp. 90–93.

[109] G. K. Wallace, "The jpeg still picture compression standard." *Commun. ACM*, vol. 34, no. 4, pp. 30–44, 1991.

[110] D. Le Gall, "Mpeg: a video compression standard for multimedia applications," *Commun. ACM*, vol. 34, no. 4, pp. 46–58, Apr. 1991.

[111] M. Wagh and H. Ganesh, "A new algorithm for discrete cosine transform of arbitrary number of points," in *IEEE Transactions on Computers*, vol. C-29, no. 4, April 1980, pp. 269 – 277.

[112] B. Lee, "A new algorithm to compute the discrete cosine transform," in *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 32, no. 6, Dec 1984, pp. 1243–1245.

[113] Y.-H. Chan and W.-C. Siu, "A cyclic correlated structure for the realization of discrete cosine transform," in *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 2, Feb 1992, pp. 109 – 113.

[114] A. B. Atitallah, P. Kadionik, F. Ghozzi, P. Nouel, N. Masmoudi, and H. Levi, "An fpga implementation of hw/sw codesign architecture for h.263 video coding," *AEU - International Journal of Electronics and Communications*, vol. 61, no. 9, pp. 605–620, 2007.

[115] Y.-P. Lee, T.-H. Chen, L.-G. Chen, M.-J. Chen, and C.-W. Ku, "A cost-effective architecture for 8x8 two-dimensional dct/idct using direct method," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 3, Jun 1997, pp. 459 – 467.

[116] W.-H. Chen, C. Smith, and S. Fralick, "A fast computational algorithm for the discrete cosine transform," in *IEEE Transactions on Communications*, vol. 25, no. 9, Sep 1977, pp. 1004 – 1009.

[117] C. Loeffler, A. Ligtenberg, and G. Moschytz, "Practical fast 1-d dct algorithms with 11 multiplications," in *International Conference on Acoustics, Speech, and Signal Processing,ICASSP-89.*, vol. 2, May 1989, pp. 988 – 991.

[118] V. Eijndhoven and F. W. Sijstermans, "Data processing device and method of computing the costine transform of a matrix," Nertherland Patent WO/1999/048 025, 1999.

[119] C.-Y. Lu and K.-A. Wen, "On the design of selective coefficient dct module," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 2, Apr 1998, pp. 143 – 146.

[120] *Accuracy requirements for implementation of integer-output 8x8 inverse discrete cosine transform*, ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) Std. ISO/IEC 23 002-1.

[121] N. Jayant, J. Johnston, and R. Safranek, "Signal compression based on models of human perception," in *Proceedings of the IEEE*, vol. 81, no. 10, Oct 1993, pp. 1385 – 1422.

[122] ITU, *ISO/IEC 10918-1 : 1993(E) CCIT Recommendation T.81*, Std., 1993. [Online]. Available: http://www.w3.org/Graphics/JPEG/itu-t81.pdf

[123] C. Lucarz and M. Mattavelli, "Dataflow/actor-oriented language for the design of complex signal processing systems," in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Brussels, Belgium, 2008.

[124] [Online]. Available: http://sourceforge.net/projects/orc-apps/